Evaluating Pre-trained Language Models for Repairing API Misuses

TING ZHANG, Singapore Management University, Singapore
IVANA CLAIRINE IRSAN, Singapore Management University, Singapore
FERDIAN THUNG, Singapore Management University, Singapore
DAVID LO, Singapore Management University, Singapore
ASANKHAYA SHARMA, Singapore Management University, Singapore
LINGXIAO JIANG, Singapore Management University, Singapore

API misuses often lead to software bugs, crashes, and vulnerabilities. While several API misuse detectors have been proposed, there are no automatic repair tools specifically designed for this purpose. In a recent study, test-suite-based automatic program repair (APR) tools were found to be ineffective in repairing API misuses. Still, since the study focused on non-learning-aided APR tools, it remains unknown whether learning-aided APR tools are capable of fixing API misuses. In recent years, pre-trained language models (PLMs) have succeeded greatly in many natural language processing tasks. There is a rising interest in applying PLMs to APR. However, there has not been any study that investigates the effectiveness of PLMs in repairing API misuse.

To fill this gap, we conduct a comprehensive empirical study on 11 learning-aided APR tools, which include 9 of the state-of-the-art general-purpose PLMs and two APR tools. We evaluate these models with an API-misuse repair dataset, consisting of two variants. Our results show that PLMs perform better than the studied APR tools in repairing API misuses. Among the 9 pre-trained models tested, CodeT5 is the best performer in the exact match. We also offer insights and potential exploration directions for future research.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**.

Additional Key Words and Phrases: empirical studies, API misuse, program repair, pre-trained language models

1 INTRODUCTION

Application Programming Interfaces (APIs) are widely used in many types of software systems, such as web applications [4] and mobile applications [18]. An API misuse refers to the use of an API that violates explicit or implicit *usage constraints* of the API [2, 3, 26]. API misuses are a prevalent issue in software development, as it is not always easy to use APIs correctly. Misuse of APIs can lead to software bugs, crashes, and security vulnerabilities [15, 76]. For instance, misuse of APIs of Secure Sockets Layer implementations (such as JSSE, OpenSSL, and GnuTLS) can lead to man-in-the-middle attacks [14]. According to a recent study [15], 17% of the bug-fixing commits are related to API misuses. To alleviate this problem, in recent years, many API misuse detection

Authors' addresses: Ting Zhang, Singapore Management University, Singapore, tingzhang.2019@phdcs.smu.edu.sg; Ivana Clairine Irsan, Singapore Management University, Singapore, ivanairsan@smu.edu.sg; Ferdian Thung, Singapore Management University, Singapore, ferdianthung@smu.edu.sg; David Lo, Singapore Management University, Singapore, davidlo@smu.edu.sg; Asankhaya Sharma, Singapore Management University, Singapore, asankhayas@smu.edu.sg; Lingxiao Jiang, Singapore Management University, Singapore, lxjiang@smu.edu.sg.

0004-5411/2023/8-ART1 $15.00
https://doi.org/XXXXXXX.XXXXXXX

approaches have been proposed [25, 37, 76]. Despite these recent efforts, API misuses remain prevalent [31, 80].

While several approaches have been proposed to detect API misuse automatically [25, 37, 76], substantial extra effort is needed to fix API misuses manually. It is desirable to have an automatic approach that can help developers repair API misuses. Unfortunately, despite its importance, there is a lack of research in developing specialized techniques for repairing API misuse. Kechagia et al. [26] made the most recent and relevant endeavor: they conducted an empirical study on 14 Java test-suite-based APR techniques regarding their ability to repair API misuses. Test-suite-based APR techniques take an input of the buggy program and a test suite where at least one failing test case is related to the bug and then generate a patch that passes the test suite [70]. They found that, while most of the patches generated by APR techniques are plausible, only a few of them are semantically correct when compared to the patches written by developers. However, their evaluation excluded *learning-aided* APR techniques, which require extra data for learning fixing patterns. Thus, how learning-aided APR techniques perform in repairing API misuses remains unknown.

Learning-aided APR usually considers the bug-fixing task as a neural machine translation (NMT) problem and attempts to tackle it by Sequence-to-Sequence (Seq2Seq) learning [79]. These techniques attempt to learn from historical fixes to repair bugs. As the state-of-the-art Seq2Seq architecture, the Transformer model [64] has received much attention in software engineering lately, particularly *general-purpose* pre-trained language models of code (PLMs) such as CodeBERT [11], CodeGPT [42], and CodeT5 [66]. These models have been extensively applied to solve various code understanding and generation tasks, including defect detection, code summarization, and code translation [65, 77]. They have also effectively repaired general program errors [69, 75]. In addition, with increased access to historical fixes, *specialized* learning-aided APR techniques [28, 41] are becoming more popular. The last couple of years have witnessed the boost of learning-aided-based APR techniques, such as SequenceR [6], CoCoNut [43], and CURE [22]. However, no attempt has been made to apply either general-purpose PLMs or specialized learning-aided APR techniques to repair API misuse bugs. Note that we consider both general-purpose PLMs and specialized learning-aided APR techniques as learning-aided APR techniques in this paper.

Moreover, the absence of benchmarks that are specifically designed for repairing API misuse bugs hinders research progress. The newly proposed APR techniques are usually evaluated on general program error benchmarks, such as Defects4J [23], QuixBugs [38] and BEARS [44]. Although there are some API misuse bugs in these benchmarks [26], they are not the main focus of these benchmarks. Thus, it lacks emphasis on the evaluation and discussion of repairing API misuse bugs. Recently, Kechagia et al. [26] shared a test-suite-based API misuse repair dataset (APIRepBench), which contains 101 API misuses. However, learning-aided APR techniques typically require a substantial amount of historical fixes for training. Thus, the limited number of API misuses in APIRepBench is insufficient to train and evaluate the performance of learning-aided APR techniques. Recently, Li et al. [34] analyzed API misuses in the wild by extracting API misuses based on 528,546 historical bug-fixing commits from GitHub (from 2011 to 2018). These commits make it possible to evaluate the performance of learning-aided APR techniques to their full potential.

In this work, we derive an API misuse repair dataset from the dataset introduced by Li et al. [34], by removing noises in the original dataset and extracting the buggy and fixed method pairs. The dataset contains 118,490 such method pairs. We also prepare a subset of the dataset that contains only one-line bugs and call it *single-line data*. We call the full dataset as *complete data*. We then evaluate the performance of learning-aided APR techniques on this dataset. To examine whether state-of-the-art PLMs are capable of fixing API misuses in methods, we compare the performance of 9 PLMs on the *complete data* (**RQ1**). We also investigate how PLMs and specific APR techniques

perform in fixing API misuses in methods with single-line changes, i.e., on the *single-line data* (**RQ2**).

To the best of our knowledge, we are the first to examine the performance of learning-aided APR techniques on API misuse repair. Our results show that PLMs are more capable of repairing API misuse than the studied APR techniques. Among the 9 PLMs tested, CodeT5 was the best performer. On the *complete data*, CodeT5 achieves an exact match ratio of 3.01. On the *single-line data*, CodeT5 achieves an exact match ratio of 8.86.

Our contributions can be summarized as follows:

- We provide a benchmark, which contains 118,490 method pairs on *complete data* and 54,510 method pairs on *single-line data*. For each method pair, the first method contains API misuse(s), and the second contains the corrected API misuse(s).
- We evaluated a total of 11 learning-aided program repair techniques on the benchmark.
- We provide implications and insights on the learning-aided APR techniques' ability to repair API misuse.

The rest of the paper is organized as follows. We outline the background in Section 2. We elaborate on the details of the experimental setting in Section 3 and results in Section 4. We discuss the implications of our work and threats to validity in Section 5. In Section 6, we list the related works. We finally conclude our work and list the potential future work in Section 7.

2 BACKGROUND

2.1 Automatic Program Repair

Automatic Program Repair (APR) aims to fix buggy programs with less manual effort. It mainly consists of two steps: (1) conducting fault localization to detect the bug and (2) generating the bug fixes. They are several ways to group the current APR techniques. We follow the prior works [13, 26] to categorize APR techniques into three groups: (1) *heuristic-based repair* techniques, (2) *constraint-based repair* techniques, and (3) *learning-aided repair* techniques.

Heuristic-based repair technique is also known as *generate-and-validate repair* technique [67]. In this category, APR is treated as a search problem. The approaches apply heuristic strategies to explore the search space and validate the candidate patches exhaustively. For example, GenProg [29] randomly selects a set of candidate patches as the initial population and then applies genetic programming to generate patches. In the patch validation stage, each patch will be validated against a test suite to compute the fitness. RSRepair [55] replaces the genetic programming in GenProg with a random search. This category also includes *template-based repair* techniques since both types work similarly [40]. For example, TBar [39] combines various fix patterns collected from previous studies. After a fix pattern is selected to repair a bug, a patch is generated and validated by a test suite. Heuristic-based repair techniques are usually restricted by the pre-defined heuristics, which may either be insufficient to cover all the possible patches or too exhaustive to be efficiently explored [13].

Constraint-based repair techniques consider APR as a constraint-solving problem. Specifically, they formulate the requirement to pass all the test cases as a set of constraints and then solve them to generate the patches [46, 51, 72]. For example, SemFix [51] infers constraints by performing symbolic executions on the supplied test cases. It then looks for concrete expressions that enable the program to pass the test cases by Satisfiability Modulo Theories (SMT) solvers. Nopol [72] uses angelic debugging to identify conditions in buggy Java programs that, if changed, could allow the program to pass the test suite. It then employs an SMT solver to synthesize repairs for these conditions [41]. Constraint-based repair techniques suffer from the scalability problem since

they usually need to conduct a heavy symbolic execution to collect constraints, and solving these constraints can be time-consuming [13].

In this work, we focus on the last category, i.e., *learning-aided repair* techniques, which leverages the previously generated bug fixes. Usually, a machine-learning or deep-learning model would be utilized to learn the bug-fixing patterns from large corpora of code [63]. Existing learning-aided repair techniques generally treat APR as an NMT problem, which translates the buggy code into the bug fixes [6, 22, 43, 63]. However, the granularity of the bug fixes is different among approaches. Some approaches work on the method level. For instance, Tufano et al. [63] evaluate an Encoder-Decoder model using the buggy method as the input and the fixed method as the output. In addition, most PLMs that have been evaluated on program repair also use the same dataset [1, 66]. Other approaches work on the line-level [22, 35, 43, 75, 81]. CoCoNut [43] takes an input of buggy lines and the buggy method as the context, and it outputs the fixed lines. We will describe more about the techniques that we evaluate in our study in Section 3.3.

2.2 API Misuses

An API *usage* can be categorized into two types, i.e., directly calling API methods or instantiating objects from API classes [50]. API *misuses* are as violations of (implicit) usage constraints of APIs [3]. Recent years have witnessed a number of API-misuse detectors [24, 25, 37, 76]. Generally speaking, there are three types of API misuse detection approaches, i.e., static detectors, which detect API misuses by statically analyzing the source code or binary code [48]; dynamic detectors, which instead detect API misuses by dynamic analysis [53]; and the third one combines mining with the static detector [54].

Regardless of the program analysis technique, existing techniques either (1) require API specification: They consider the violation of a specification as an API misuse. The drawback is that the specification is usually incomplete and hard to obtain [37]; or (2) do not require API specification: Given a large-scale code corpus, they consider the majority usage pattern valid. The use of an API is considered to be a misuse if it does not follow the majority usage pattern. The drawback is that this assumption does not always hold. For instance, some APIs are rarely used, and the majority usage pattern does not represent valid usage [24].

2.3 API Misuse Benchmark

Recently, several API-misuse benchmarks have been proposed. Amann et al. [3] proposed the first benchmark named MuBench for API misuse detection. MuBench encompasses 89 API misuses. Among them, 77 API misuses are from real-world projects, and the remaining 12 misuses are from the survey they conducted. The 77 API misuses are collected from (1) existing bug datasets, i.e., BugClassify [19], Defects4J [23], and QACrashFix [12]; (2) bug-fixing changes from projects on SourceForge and GitHub for misuses of Java Cryptography Extension (JCE) APIs. MuBench is the first benchmark that has been used to evaluate API-misuse detectors.

APIRepBench [26] is the first benchmark that can be used for API misuse repair. It contains 101 API misuses from 29 Java projects. This benchmark is derived from three existing bug benchmarks, i.e., MuBench [2], BEARS [44], and Bugs.jar [59]. It only contains API misuses that belong to the *missing* category, such as missing method calls, missing null checks, and missing exception handling. Due to the limited size of this benchmark, it is insufficient to train and evaluate the learning-aided APR techniques.

More recently, Li et al. [34] created a large-scale API misuse repair benchmark, which contains 528,546 bug-fixing commits of Java projects from 2011 to 2018. They extracted fine-grained edit operations on the Abstract Syntax Tree (AST) of the source code. They also classify the API misuses into different categories, including *missing*, *redundant* and *replaced*. Based on the categories, they

```
{
  "Line":"=> 29",
  "parseTypeFail":"success",
  "Pattern":"UNKNOWN=>LocalVariable",
  "RetCheckAPI":[],
  "Type":"Insert",
  "BugDetectionTag":"[]",
  "Content":"=> int counter = 0",
  "FileName":"/home/PATH/BugDetectionProject/3Clone/NewDown2011-2017/All/V9/8530/buggy-
      version/jTowerDefense.src.model.Path.java",
  "BodyUseAPI":[],
  "Fixed commit":"96257b05bd27ed7a3d3ba55c80cf40de6aebb015",
  "Url":"https://api.github.com/repos/bernhardfritz/jTowerDefense",
  "Date":"2014-04-02T19:51:51Z"
}
```


Fig. 1. An example of a data point in *Li et al.'s data*.

extracted frequent API misuse patterns. Since this dataset is the largest and most recent, we derive a dataset that contains method pairs from it. We describe the detail in Section 3.2.

3 EXPERIMENTAL SETUP

3.1 Research Questions

In this study, our primary objective is to investigate the effectiveness of learning-aided repair methods in repairing API misuses, particularly when employing general-purpose PLMs.

With this goal in mind, we aim to answer the following two Research Questions (RQs).

- **RQ1:** *How effective are PLMs for fixing API misuses in methods?* With the first RQ, we aim to examine the effectiveness of PLMs regarding their capability to repair API misuses in the method granularity. We are interested in investigating this RQ to ascertain the possibility of integrating PLM-based learning-aided APR techniques directly into real development.
- **RQ2:** *How do PLMs and specific APR techniques perform in fixing API misuses in methods with single-line changes?* With the second RQ, we investigate the effectiveness of learning-aided APR techniques in repairing API misuses. We run two state-of-the-art APR techniques, i.e., SequenceR [6], Recoder [81]. Since SequenceR and Recoder are designed to repair single-line and single-hunk bugs, respectively, we further derive a smaller dataset (named *single-line data*) containing only single-line bugs, which can be fixed by modifying one line. Thus, we train/fine-tune and evaluate all the models on *single-line data*.

3.2 Dataset

In this section, we describe how we build the benchmark dataset. We leverage the dataset published by Li et al. [34] (*Li et al.'s data*). Figure 1 shows one data point in *Li et al.'s data*. We exclude the data points where (1) buggy line or fixed line is missing in `Line`, (2) `parseTypeFail` is not equal to `success`, or (3) the `Pattern` contains `UNKNOWN`. In this example, the buggy line is empty, and the `Pattern` contains `UNKNOWN`; thus, we exclude this data point from our dataset. Although the dataset contains the commits that fixed API misuses (`Fixed commit`), it lacks the commits that contain the bugs. We refer to these two types of commits as *fixed* commits, and *buggy* commits, respectively. We extracted buggy commits by referring to the parent commits of the fixed commits. Next, we

Table 1. Final dataset: the number of method pairs in each set and the average number of tokens in the src (buggy method) and tgt (fixed method).

Variant	Train.	Valid.	Test	Avg. # Tokens	
				SRC	TGT
complete data	94,792	11,849	11,849	112	115
single-line data	43,608	5,451	5,451	90	90

Table 2. Comparison of the selected *general-purpose* PLMs.

Model	Arch.	# of Parameters
CodeBERT [11]	Enc.	125M
GraphCodeBERT [17]	Enc.	125M
CodeGPT [42]	Dec.	124M
PolyCoder-160M [71]	Dec.	160M
PolyCoder-0.4B [71]	Dec.	400M
CodeTrans [10]	Enc. & Dec.	223M
PLBART [1]	Enc. & Dec.	140M
CodeT5 [66]	Enc. & Dec.	223M
UniXCoder [16]	Enc. & Dec.	125M

downloaded the Java files of the two versions. We used JavaParser [1] to remove comments and parse the Java files. We removed Java files that JavaParser cannot parse. Similar to the prior work [63], we focus on the method granularity in this work. Based on the `Line` information provided in Li et al.'s dataset, we extracted the buggy and fixed method pairs. We further removed the duplicate method pairs, which may come from different forks while having the same base repository. Finally, we shuffled the dataset and split the dataset into training, validation, and testing sets with a ratio of 8:1:1, as the *complete data*. To get the *single-line data*, we filter the bugs which only involve one-line change. We also split them into training, validation, and testing with the same ratio. For PLMs, the training and validation data are used for fine-tuning the models. Table 1 shows the statistics of our dataset, i.e., *complete data* and *single-line data*.

3.3 Selected Approaches

Learning-aided APR approach takes a code snippet with API misuse(s) as the input sequence, and generates a fixed version of this code snippet that does not contain API misuse(s) as the output sequence. We consider two types of learning-aided APR approaches, i.e., *general-purpose* pre-trained models of code, which can be used to solve several programming understanding and generation tasks [42, 77], and the *specialized techniques*, which are proposed to repair program errors.

General-purpose Pre-trained Models of Code. PLMs mainly differ in architecture and pre-training tasks. We include the PLMs that adopt three types of architecture, i.e., encoder, decoder, and encoder-decoder. For each type of architecture, we choose state-of-the-art models that have demonstrated effectiveness in a recent study on programming understanding and programming generation tasks [77]. We also consider the more recent PLMs, i.e., PolyCoder [71] and UniXCoder [16]. Table 2 shows the considered PLMs' architecture and the number of parameters.

[1] https://github.com/javaparser/javaparser

- **CodeBERT [11]** is a bi-modal pre-trained model for programming language (PL) and natural language (NL). It has been pre-trained with a hybrid objective function, including standard masked language modeling (MLM) [27] and replaced token dection [7]. Simply put, the MLM objective is to predict the original tokens which are masked out. The replaced token detection objective predicts whether a token is an original token or not. In the pre-training phase, the input is the concatenation of NL text and code in a certain PL with a special token to separate them. CodeBERT considers both NL and PL as a sequence of words. Specifically, the pre-training corpus of CodeBERT is a recent dataset CodeSearchNet [21], which contains 2.1M bimodal data points and 6.4M unimodal data points.
- **GraphCodeBERT [17]** considers the inherent structure of code. In the pre-training stage, it uses data flow, a semantic-level structure of code that encodes the relationship between variables. Other than MLM, GraphCodeBERT also adopts two structure-aware pre-training tasks: one is data flow edge prediction, which aims to learn representation from code structure; the other is to align representations between source code and code structure. GraphCodeBERT was pre-trained on the CodeSearchNet dataset [21], which contains 2.4M functions of six programming languages paired with natural language documents.
- **CodeGPT [42]** has the same model architecture and pre-training objective as GPT-2 [56]. Specifically, CodeGPT has 12 layers of Transformer decoders. GPT-2 is trained with a simple objective: predicting the next token one by one, which is conditioned on its previous tokens and itself. This is also called auto-regressive language modeling. The one we evaluated in our work is the *CodeGPT-adapted* variant, which uses GPT-2 as the starting point and is further pre-trained in the 1.6M Java methods from CodeSearchNet dataset [21]. It has the same vocabulary and natural language understanding ability as the original GPT-2.
- **PolyCoder [71]** is also based on GPT-2 structure [56] and was pre-trained on the dataset collected by its authors. After deduplication and filtering, its pre-training dataset contains 24.1M files and 254GB of data across 12 PLs. In the original work, the authors trained three models in different sizes, with 2.7 billion, 400 million, and 160 million parameters. Considering our budget, in our work, we choose the last two models, which have 160M and 400M parameters, respectively. We leave the evaluation of a larger variant, i.e., the 2.7B-parameter model, for future work.
- **CodeTrans [10]** is based on the T5 architecture [57]. In the pre-training stage, CodeTrans involved six different corpora for unlabeled datasets, which cover 9 PLs and English text. In total, it has around 40 million samples. It applies the span corruption task with a corruption rate of 15% as the pre-training task. It corrupts the input sequence by masking a span of tokens, and then the model is trained to predict the masked spans.
- **PLBART [1]** is a bidirectional and autoregressive Trans- former pre-trained on unlabeled data across PL and NL. PLBART uses the same architecture as BART_{base} [32]: it has 6 layers of an encoder and 6 layers of a decoder. Similar to CodeTrans, PLBART also uses denoising Seq2Seq pre-training: the model learns to reconstruct an input text that is corrupted by a noise function. Specifically, PLBART adopts three noising strategies: token masking, deletion, and infilling. PLBART has been pre-trained on a large collection of Java and Python functions and their NL descriptions from GitHub and Stack Overflow.
- **CodeT5 [66]** also builds upon the T5 architecture [57], while it considers the token type information in code. Similarly, CodeT5 employs a denoising sequence-to-sequence pre-training task. Moreover, CodeT5 leverages the code semantics conveyed by the developer-assigned identifiers. The model is also pre-trained with two identifier-related tasks. The first task is *identifier tagging*, and aims to distinguish whether the code token is an identifier or not. The other task is *masked*

identifier prediction, which corrupts the input sequence by masking all the identifiers in the PL segment and employs a sentinel token for all occurrences of one specific identifier.

- **UniXCoder [16]** is a unified cross-modal pre-trained model for PL. Its pre-training tasks are MLM, unidirectional language modeling, and the denoising objective. After these three pre-training tasks, to learn the semantic embedding, UniXCoder proposes two pre-training tasks, i.e., multi-modal contrastive learning and cross-modal generation. For multi-modal contrastive learning, positive examples are the same input with different hidden dropout masks, and negative examples are other representations in the same batch. For the cross-modal generation, the model needs to generate a comment describing the function of the code. UniXCoder supports three modes: *encoder*, *decoder*, and *encoder-decoder*. We use the encoder-decoder mode of UniXCoder.

Specialized Techniques. To select specialized APR techniques, we mainly rely on the latest live review about automatic program repair [47], which was submitted on 9 Aug 2022. We mainly focus on the general data-driven APR approaches proposed in the last three years (2020 - 2022). Thus, we do not consider the domain-special APR techniques, such as APR for null pointer error [30] and APR for concurrency errors [33]. The criteria to be selected for inclusion in our study are (1) a new APR approach has been proposed, (2) the source code is publicly available, and (3) the authors provide an easy-to-follow guide to reproduce their work so that we can adapt the proposed approach to a new dataset. Note that it is not a trivial task to replicate these APR techniques, as they are often implemented in different deep-learning libraries and require different dependencies. Existing APR techniques usually compare with each other in Defects4J [23]; thus, they often directly cite the results of prior reported results instead of re-running the experiments [22, 39, 43, 69, 81]. Furthermore, given the large size of our dataset, it is infeasible to run test suites for each potential fix constantly. Thus, we exclude APR techniques that rely on test suites (either in the training or inference stage), e.g., RewardRepair [74] and SelfAPR [73] involve test cases in the training stage, and DEAR [35] requires test cases in the inference stage. Given we already have a group of PLMs, we excluded the APR techniques which rely on PLMs, such as CURE [22]. In the end, we choose one Seq2Seq model, i.e., SequenceR [6], and one tree-based model, i.e., Recoder [81].

- **SequenceR [6]** is based on Seq2Seq learning and it adopts copy mechanism [60] to overcome the unlimited vocabulary issue in source code. It is specifically designed to solve one-line patch generation task, i.e., the bug can be fixed by replacing a single line.
- **Recoder [81]** is built based on encoder-decoder architecture. It uses a syntax-guided edit decoder with placeholder generation, aiming to generate a sequence of edits rather than a new statement. The usage of this decoder tackles the problem of inefficient representation of small edits. Moreover, it also enables Recoder to generate project-specific identifiers by leveraging a neural network for placeholder generation. Recoder leverages AST in the framework. It first transforms the raw method into an AST, which is then modified and used to extract the rules. The generated rule will be used in the training and inference phase.

3.4 Implementation

PLM. We implement PLMs with Hugging Face `Transformers` library [68]. We run each model for 30 epochs under each setting. However, if the loss in the validation set does not decrease for 5 epochs, the early-stopping strategy would be triggered. We used the model which has the smallest loss on the validation set as the final model. The used hyper-parameters are available in our replication package [2]. We run the experiments on a machine with 4 NVIDIA RTX A5000 GPUs and the AMD EPYC 7643 48-Core Processor.

[2] https://anonymous.4open.science/r/TOSEM-API-Misuse

APR techniques. We set a beam size of five for both approaches. For SequenceR, we keep the Top 1 prediction result as the PLMs. We used the model which achieves the highest accuracy on the validation set as the final model. For Recoder, we set the batch size to 16, and the epoch to 30 for the training phase. In the inferring phase, we keep the Top 1 prediction result as the fix for the buggy code. As for other hyper-parameters, we used the default values as provided in the replication package.

To fairly compare the performance of PLMs and the studied APR techniques, we give the same information to all the models, i.e., the input is the buggy method. Even so, the specific input format may vary across different models. Based on the original design, for SequenceR [6], we provide `<START_BUG>` and `<END_BUG>` labels to explicitly separate the buggy lines and the context in the buggy method. As for Recoder [81], we preprocess the raw buggy method with a script provided in Recoder's replication package to prepare the input data. In general, it takes the raw buggy method as input in the training phase. For the inference phase, we need to provide the location of the buggy line along with the buggy method. On the other hand, the rest models were given the raw buggy method as the input.

3.5 Evaluation Metrics

Following prior work [30, 66], we use the Exact Match (EM), BLEU [52], and CodeBLEU [58] to measure the quality of the generated fixed code. We consider EM as the critical metric, as it is the strictest one. We also consider BLEU, since it is the widely-used metric for NMT. On the other hand, CodeBLEU considers the unique syntax and semantics of source code. Moreover, compared with BLEU and EM, it shows a better correlation with the programmer-assigned scores [58]. We describe each metric as follows.

EM (Exact Match) is defined as the percentage of the generated fixed code that *exactly matches* the reference fixed code. It is a strict metric that can exclude some successful repairs. EM can be considered as the lower bound since the different programs can be semantically the same but are written differently from the developer-written code.

BLEU-4, calculates the percentage of 4-gram overlap between the *reference* fix code and the *candidate* code. For simplicity, we refer it as *BLEU* in our work. Candidate fixed code represents the code generated by models, while reference fixed code represents the ground-truth code written by developers. BLEU is defined as Equation 1:

$$\text{BLEU} = \frac{\sum_{P \in C} \sum_{4-gram \in P} \text{Count}_{\text{matched}}(\text{4-gram})}{\sum_{P \in C} \sum_{4-gram \in P} \text{Count}(\text{4-gram})} \tag{1}$$

, where P refers to each candidate, fix code generated by the model, C refers to the whole candidate code. Since it was proposed for evaluating NL, it neglects the syntax and semantics included in the source code.

To remedy the drawback of applying BLEU in source code tokens, we include the newly introduced metric *CodeBLEU*. CodeBLEU considers both syntactic match and semantic match by injecting code syntax via AST and code semantics via data flow. CodeBLEU is defined in Equation 2:

$$\begin{aligned} \text{CodeBLEU} &= \alpha \times \text{BLEU} + \beta \times \text{BLEU}_{\text{weight}} \\ &+ \gamma \times \text{Match}_{\text{ast}} + \delta \times \text{Match}_{df} \end{aligned} \tag{2}$$

where $BLEU_{weight}$ is the weighted n-gram match, $Match_{ast}$ is the syntactic AST match, $Match_{df}$ is the semantic data-flow match. Both $BLEU$ and $BLEU_{weight}$ work in sequence-level matching, while the latter one considers the keywords in each PL. $Match_{ast}$ calculates the accuracy by comparing the sub-trees from both candidate and reference code. $Match_{df}$ computes the semantic data-flow

Table 3. Results of PLMs in repairing API misuse on *complete data*. The best performer in each group is in bold.

Model	BLEU	CodeBLEU	EM
Encoder-based			
CodeBERT [11]	**51.7**	**58.22**	1
GraphCodeBERT [17]	49.21	56.25	**1.55**
Decoder-based			
CodeGPT [42]	50.85	57.57	2.02
PolyCoder-160M [71]	54.58	60.36	2.77
PolyCoder-0.4B [71]	**54.88**	**60.62**	**2.96**
Encoder-decoder-based			
CodeTrans [10]	53.12	59.24	2.68
PLBART [1]	**54.9**	60.15	1.34
CodeT5 [66]	54.08	60	**3.01**
UniXCoder [16]	54.76	**60.36**	2.17

match score between the candidate and reference code. In our work, we use the default values of α, β, γ, and δ as 0.25.

4 RESULTS

4.1 RQ1: How effective are PLMs for fixing API misuses in methods?

Quantitative Analysis. Table 3 shows the performance of the three groups of fine-tuned PLMs. We can find that decoder-based and encoder-decoder-based PLMs generally perform better than encoder-based PLMs, except that PLBART achieves a lower EM ratio than GraphCodeBERT. Based on the task characteristics, the decoder-based and the encoder-decoder-based PLMs are more suitable for APR. Specifically, the pre-training tasks adopted by the encoder-decoder group resemble our API misuse repair task the most. It suggests that the Seq2Seq pre-training can benefit the downstream Seq2Seq task, where in our case, the downstream task is repairing API misuse.

The best-performing encoder-decoder-based model, CodeT5, performs similarly to the best-performing decoder-based model, i.e., PolyCoder-0.4B. Still, CodeT5 outperforms PolyCoder-0.4B by 1.7% in terms of EM. Besides, PolyCoder-0.4B and CodeT5 achieve similar BLEU and CodeBLEU scores. Moreover, all the four encoder-decoder-based models achieve similar BLEU and CodeBLEU scores. This suggests that the repairs generated by encoder-decoder-based PLMs are likely to be syntactically correct or semantically similar to the developer-written repair. The gap between CodeT5 and the best-performing encoder-based model, i.e., GraphCodeBERT, is more pronounced. According to our main metric, i.e, EM, CodeT5 outperforms GraphCodeBERT by 94.2%. For the other two metrics, CodeT5 also achieves higher values.

We also compare the performance of PLMs within the same group. The result shows that PolyCoder can achieve a higher EM than CodeGPT. It suggests that the larger the pre-training corpora, the model tends to perform better. PolyCoder and CodeGPT are based on the GPT-2 architecture, while PolyCoder has been pre-trained in a larger corpus than CodeGPT. Similarly, the better performance of PolyCoder-0.4B over PolyCoder-160M indicates that, with the same model architecture and the same pre-training task, the one having more parameters tends to achieve better performance.

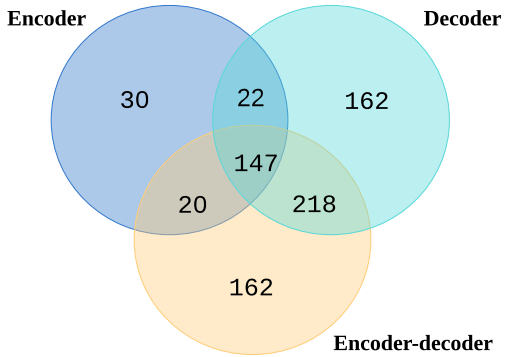

Fig. 2. The number of successful repairs produced by different groups of PLMs in the *complete data*.

We merge the successful repair produced by the PLMs inside the same group. We consider the repair to be *successful* if the generated repair is exactly the same as the developer-written repair. Figure 5 shows the overlap among the successful repairs produced by different groups of PLMs. We can find that fine-tuned decoder-based and encoder-decoder-based models can produce more successful repairs than encoder-based models. We further analyze the methods that each PLM group has successfully repaired. All three groups of PLMs successfully repair the same 147 buggy methods. We find that each group of PLM can successfully generate the correct repairs for some buggy methods that the other two groups cannot. We also observe that any two groups of PLMs have common methods that they both successfully repair, but the remaining group cannot. In total, all the PLMs can repair 761 buggy methods, which accounts for 6.42% of all the method pairs.

Qualitative Analysis. To understand the reason behind the performance difference between these groups of PLMs, we analyze the generated repairs. Specifically, we performed a qualitative analysis to understand when the APR techniques can fix the API misuse, and when they cannot. To understand why they can fix API misuses, we investigate the common successful repairs produced by the three groups of PLMs. We manually check the successful repairs by all three groups of PLMs (i.e., 147 repairs). Two authors independently categorized all 147 buggy-fixed method pairs into three mutually exclusive categories. These categories were based on whether the fixes could be generated by: (1) Understanding method semantics, (2) Changing standard Java APIs, or (3) Others, which encompassed fixes requiring information beyond the buggy method. The authors then discussed and resolved any disagreements. The resulting categories consisted of 77 pairs (52.4%), 13 pairs (8.8%), and 57 pairs (38.8%), respectively. We show two examples in Figure 3, where lines starting with '-' are the buggy lines, and those starting with '+' are the correct fixes written by developers. We have the following findings.

(1) *PLMs can understand the method semantics.* We identify several fixes related to correcting the use of API calls. When the API name conveys meaningful information or some existing statements as the context, PLMs can identify statements that are inconsistent with the method context. Consider Example 1 in Figure 3, inside the definition of method `clonePeriodAfterMidnight`, there are four statements. Line 3 sets the start date. Naturally, Line 4 should set the end date. The API call was correct, i.e., `result.setEndDate()`, but the argument, which is filled with another API call was wrong: `getStartDate()` is called instead of `getEndDate()`. Based on the method name and the context of the method, PLMs can learn to fix this type of bug.

Example 1: Inferred based on the semantics of the identifier

```
1  private Period clonePeriodAfterMidnight(Period source) {
2    Period result = new Period();
3    result.setStartDate(source.getStartDate().plusDays(1));
4  -  result.setEndDate(source.getStartDate().plusDays(1));
5  +  result.setEndDate(source.getEndDate().plusDays(1));
6    return result;
7  }
```

Example 2: Standard Java API

```
1 public boolean containsKey(Object key) {
2 -  return this.parametersHashTable.contains(key);
3 +  return this.parametersHashTable.containsKey(key);
4 }
```

Fig. 3. Common successful fixes generated by PLMs

(2) *PLMs can repair the misuse of standard Java APIs.* We find that PLMs can repair the APIs in the standard Java libraries. For Example 2 in Figure 3, both `contains` and `containsKey` are correct APIs in `HashTable`. However, the arguments are different. `contains` checks whether some key maps into the specified *value* in this HashTable; while `containsKey` checks whether a specified object is a *key* in this HashTable. Based on the parameter name, the function aims to check whether the key exists.

To understand why PLMs fail to generate the correct repair, we randomly sampled 100 methods that all PLMs fail to repair. The aim is to determine the challenges of repairing API misuse with PLMs. We found several cases where the current PLMs fail.

(1) *When failing to consider the impact of its changes.* In some cases, PLMs provide a partial fix. Take Example 3 in Figure 4: in the buggy method, if the parameter `value` is not null, the `value` would be set as an empty string. As a result, the argument of the API call in Line 7 would always be the same. It is certainly not the intention of this method. The fix generated by CodeT5 makes more sense: if the `value` is `null`, return `false`; otherwise, make the API call in Line 7. However, it neglects the fact that the String casting only works if the Object is an instance of String. Since `value` can be any Object, using `(String)` is erroneous. The fix provided by a developer handles this issue.

(2) *When the intention is unclear.* In some cases, without knowing the developer's intention, both the buggy and fixed versions may look bug-free. As shown by Example 4 in Figure 4, the buggy method and the fixed method differ in the internal class property being called, i.e., either a button or a view. Without knowing the exact intention and what `ptt` refers to, it is nearly impossible to fix this kind of bug. Given the *buggy* version can be considered correct, the generated repair by CodeT5 and PolyCoder are the same as the buggy version. On the other hand, CodeBERT replaces the parameter with `true`.

Answer to RQ1: PLMs are effective in fixing API misuses in methods. Specifically, decoder-based and encoder-decoder-based PLMs perform better than encoder-based PLMs.

Example 3: Partial fix (Developer-written Fix)

```
 @Override
public boolean isValid(Object value) {
- if (value != null) {
-      value = "";
- }
+ value = value.toString();
   return mPattern.matcher((String) value).matches();
}
```

Repair generated by CodeT5

```
 @Override
public boolean isValid(Object value) {
- if (value != null) {
-     value = "";
+ if (value == null) {
+     return false;
   }
   return mPattern.matcher((String) value).matches();
}
```

Example 4: Lack of context

```
private void updatePTTConfiguration() {
- mTalkButton.setVisibility(settings.isPushToTalk() && settings.isPushToTalkButtonShown() ?
    View.VISIBLE : View.GONE);
+ pttView.setVisibility(settings.isPushToTalk() && settings.isPushToTalkButtonShown() ?
    View.VISIBLE : View.GONE);
}
```

Repair generated by CodeBERT

```
private void updatePTTConfiguration() {
- mTalkButton.setVisibility(settings.isPushToTalk() && settings.isPushToTalkButtonShown() ?
    View.VISIBLE : View.GONE);
+ mTalkButton.setVisibility(true);
}
```

Fig. 4. Wrong fixes generated by PLMs

4.2 RQ2: How do PLMs and specific APR techniques perform in fixing API misuses in methods with single-line changes?

Quantitative Analysis. Table 4 shows the comparison of PLMs against the state-of-the-art APR techniques on the *single-line data*.

Table 4 shows that fine-tuned PLMs perform better than studied APR techniques, which demonstrates the effectiveness of pre-training: all of the PLMs are pre-trained in a large corpus to capture the syntactic and semantic knowledge of source code. The best-performing model, CodeT5, outperforms Recoder by around 12 times in terms of EM. SequenceR and Recoder perform worse than PLMs, with the worst-performing CodeBERT achieving more than double the EM ratio. Regarding BLEU and CodeBLEU, it may be surprising to see that the studied APR approaches surpass the PLMs. Simply copying the buggy method as the fixed method would result in a BLEU and CodeBLEU close

Table 4. Results of PLMs and APR techniques on the *single-line data*.

Model	BLEU	CodeBLEU	EM
Encoder-based			
CodeBERT [11]	**63.11**	**66.58**	2.18
GraphCodeBERT [17]	60.5	64.17	**2.49**
Decoder-based			
CodeGPT [42]	62.21	65.9	4.22
PolyCoder-160M [71]	65.7	**68.66**	5.93
PolyCoder-0.4B [71]	**65.73**	68.65	**6.31**
Encoder-decoder-based			
CodeTrans [10]	64.43	67.47	5.65
PLBART [1]	**66.13**	**68.8**	5.1
CodeT5 [66]	65.18	68.27	**8.86**
UniXCoder [16]	66.01	68.73	4.97
SequenceR [6]	93.58	89.75	0.48
Recoder [81]	**94.05**	**93.4**	**0.72**

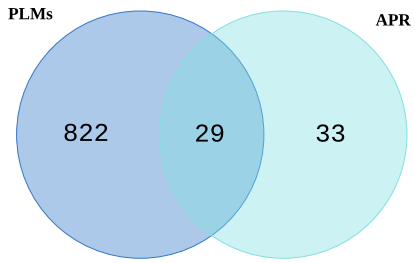

Fig. 5. The number of successful repairs produced by PLMs and APR techniques in the *single-line data*.

to 100. It is worth reminding that both selected APR approaches generate the fixed line instead of the whole method. Thus, the large portion of the fixed method would be the same as the buggy method. Therefore, it is intuitive that the APR approaches can achieve higher BLEU and CodeBLEU.

Qualitative Analysis. In this section, we performed two sets of manual checks:

1. Evaluation of the common successful fixes generated by PLMs and APR techniques. We thoroughly examined all 29 identified cases, which collectively represent the total number of fixes commonly identified across the various approaches.

2. Analysis of the characteristics of successful fixes produced solely by APR techniques. We carefully studied 33 cases where the APR tools successfully fixed the bugs, while the PLMs were unable to do so.

(1) *Common successful fixes.* The fixes relate to the method argument. For example, the fix is to change the boolean argument in a method call: the fix is either changing `true` to `false` or vice versa. We found 11 out of 29 cases (37.9%) were resolved in this manner.

(2) *Characteristic of unique successful fixes by APR techniques.* Similar to the prior finding, we found 12 out of 33 cases (36.4%) were fixed by changing the boolean argument in the method call. The difference is that most cases involve larger methods with more lines than cases in the common successful fixes. This applies not only to the boolean argument fixes. We also find other

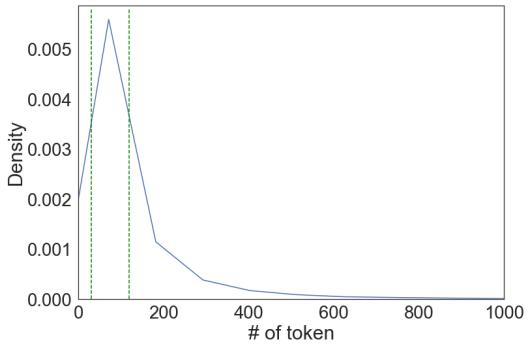

Fig. 6. Distribution of buggy methods by the number of tokens in *complete data*.

successful fixes are generally longer, which makes it challenging to localize the API misuse and therefore results in PLMs failing to produce the correct fix. Since PLMs lack explicit information on fault localization, APR techniques may be more aware of the faulty location. This suggests that an accurate bug localization may have benefited the APR techniques.

Answer to RQ2: PLMs are more effective than the studied APR techniques in fixing API misuses in methods with single-line changes.

5 DISCUSSION

5.1 Impact of the Program Length on the Model Performance

In Section 4, when we investigate the reasons why APR techniques fail to successfully produce the repairs, we noticed that the failure cases have relatively longer method lengths compared to the successful cases. Therefore, we investigate whether the length of the buggy methods affects the model performance.

From Figure 6, we notice that there is a peak before 100 tokens and a long tail that extends over 300 tokens. The first quartile for this distribution is 30 tokens, and the third quartile is 119 tokens. We consider *short* methods as buggy methods that have less or equal to 30 tokens and *long* methods as buggy methods that have more than 119 tokens. Since we want to focus on comparing the performance when the length difference of the methods is more pronounced, we only report the results with short and long methods.

Table 5 shows the results of different PLMs on both *short* and *long* method pairs. We can see that the performance of all the approaches is worse on the long method pairs compared to that on the short method pairs. This finding is consistent with the results of Tufano et al. [63]. Tufano et al. [63] found that NMT-based APR can perform better on the shorter methods. Note that our methods are longer than the dataset provided by Tufano et al.. The results from our experiments and Tufano et al. both indicate that the longer the method, the more complex the program logic is. Hence, more difficult it is to produce the correct repair.

Comparing the results of the approaches on the short method pairs with their results on the whole test set, we can see that the BLEU, CodeBLEU, and accuracy achieved by all the approaches have increased. Similarly, we find that the performance of all the approaches on the long method pairs is worse than that of the whole test set. It suggests that localizing faults in longer methods is more challenging, and it is possible longer methods have potentially more faulty locations, where more transformations are needed to generate the correct repair.

Table 5. Results on the *short* (less or equal to 30 tokens) and *long* (more than 119 tokens) method pairs in *complete data*.

	BLEU		CodeBLEU		EM	
	short	**long**	**short**	**long**	**short**	**long**
CodeBERT	**60.64**	**33.2**	**65.85**	**45.98**	2.4	0.03
GraphCodeBERT	60.22	30.3	65.56	43.52	**3.5**	**0.17**
CodeGPT	**60.03**	32.21	**65.37**	45.28	4.35	0.1
PolyCoder-160M	58.29	38.42	63.89	49.68	5.87	0.27
PolyCoder-0.4B	59.51	**38.52**	65.06	**49.75**	**6.28**	**0.34**
CodeTrans	61.12	35.32	66.33	47.45	**5.46**	0.34
PLBART	60.91	38.08	65.95	49.01	2.71	0.21
CodeT5	**61.17**	36.74	**66.39**	48.47	6.15	**0.38**
UniXCoder	60.14	**38.62**	65.65	**49.61**	4.95	0.03

5.2 Lessons Learnt

We identify several insights that we hope can inspire the development of specialized techniques for repairing API misuse. Specifically, we consider the unique challenges faced by learning-aided APR techniques in repairing API misuse.

Repairing API misuses in *long* methods is challenging. Based on the result in Section 5.1, we find that current PLMs lack the ability to repair long methods. PLMs, such as CodeBERT, can only handle input less or equal to 512 tokens by default. However, some *long* methods have more than 512 tokens. A simple truncation does not work well, as shown in Table 5. Moreover, filtering out long methods is not solving the problem. It may also be unrealistic to completely disallow developers from writing long methods. On the other hand, splitting long methods into chunks or focusing on lower granularity may help in repairing API misuses.

Integrating project-specific information can potentially improve repair performance. Like Example 4 shown in Figure 4, this type of bug is unrelated to the syntactic change, while is more related to the project information. This is a unique challenge in learning-aided APR techniques. Learning-aided techniques usually have a limitation on the input length. It is infeasible to treat the whole project information as input. More ways to integrate project information should be a future direction to explore.

Prioritize frequent and non-trivial API misuses. As shown in Section 4, it is challenging to repair API misuses. Considering various types of API misuses, it may not be possible to completely repair all of them. Some types of API misuse can be considered trivial for developers yet not easy for machines to repair. We believe it is not worth resorting to an automatic approach to repair all API misuses. Instead, automation efforts should be focused on frequently occurring API misuses. Some frequently occurring API misuses are common program errors. It would save a lot of manual work if an automatic approach could be integrated when developers are coding. Other than frequently occurring API misuses, more emphasis should be put on non-trivial API misuses, such as those that have a high impact on security or require more expertise to repair. As the next step, researchers can focus on a certain type of critical API misuse and develop a specialized tool for it.

Except for the three points mentioned above, adopting learning-aided APR techniques for API misuse also shares common challenges with test-suite-based APR techniques as presented by Kechagia et al. [26]. Here, we name a few: (1) incorrect fault localization: without correctly

locating the API misuse bugs, it is impossible to repair the misuse; (2) multiple faulty locations: multi-locations bugs are challenging to repair.

5.3 Threats to Validity

Threats to internal validity relate to the correctness of our experiments. For SequenceR and Recoder, we directly used the default settings as the original paper unless specified otherwise. For general-purpose PLMs, we implement the models based on the replication package released by Zeng et al. [77]. We believe the threats are minimal.

Threats to construct validity relate to experimental bias. Following the prior works [42, 77], we not only use the exact match and BLEU but also include the newly introduced CodeBLEU score, which is more suitable for code as the evaluation metrics. While we admit that a more rigorous way of evaluating the generated methods is to actually compile them, it is not feasible in practice to evaluate a large number of methods.

Threats to external validity relate to whether our findings can be generalized to other datasets or PLMs. In this work, we experiment with bug fixes from Java projects. The results may differ when we experiment with Python projects. However, since the pre-trained models are not specifically designed for a certain PL, we believe the threat is minimal. In the future, we plan to conduct experiments on dataset containing other PLs. Another threat to external validity is on the approaches' selection. We select two state-of-the-art APR techniques: SequenceR and Recoder. They represent two types of architecture, i.e., Seq2Seq-based and tree-based model. We experiment with 9 PLMs, and larger PLMs have been released recently (e.g., CodeT5 has released a larger version [3]). In the future, we plan to experiment with large PLMs.

6 RELATED WORK

6.1 Pre-trained Language Models for Program Repair

Given the success of PLMs in NLP tasks, researchers in software engineering are exploring their potential for use in general APR. Several studies have proposed novel APR methods built upon PLMs [22], while others investigate alternative ways to leverage them [69]. To improve APR, existing approaches often utilize additional information beyond the buggy code. For example, TFix [5] is based on the T5 model [57] and requires the error messages from error detectors such as ESLint as input to generate bug fixes. Additionally, specialized PLMs designed to boost code review progress have been proposed [36, 78], but these PLMs are meant to solve a different type of APR that requires NL reviews. Instead of traditional fine-tuning, recent works aim to leverage the pre-training objective (MLM) in PLMs and close the gap between pre-training and repair [69, 75]. Xia and Zhang [69] explore zero-shot learning for generating bug fixes by treating program repair as a cloze-style task, where the buggy line is masked and the PLM is required to fill in the missing code. They have explored various templates to mask lines. Similarly, CIRCLE [75] uses a prompt-based template to convert APR into a "fill-in-the-blank" task. In our work, we conduct a comprehensive evaluation on various PLMs with traditional fine-tuning. In the future, we aim to investigate alternative ways of applying PLMs to repair API misuse.

6.2 Domain-specific Program Repair

Other than general APR techniques considered in our work, several more studies have focused on repairing domain-specific program errors, such as concurrency errors [33], the web [45], security vulnerabilities [20], Android applications [61] and regression bugs [62].

[3] https://huggingface.co/Salesforce/codet5-large

Li et al. proposed a tool named DFix [33], which focuses on fixing timing bugs in distributed systems. Fixing distributed timing bugs has unique challenges: (1) it cannot rely on traditional synchronization primitives, such as locks and conditional variables; (2) it requires global code changes. DFix can automatically process distributed timing bug reports, analyzes the buggy system and generates patches through static program analysis. At a high level, DFix systematically generates patches that handle observed buggy timing through rollbacks or fast-forwards. At a low level, DFix uses static analysis to automatically decide where and how to observe buggy timing and where and how to conduct rollback or fast-forward.

Domain-specific program repair techniques usually need to consider the uniqueness of the domain. For instance, Mahajan et al. [45] proposed a new approach that can automatically generate CSS patches that improves the mobile-friendliness of a web page. Existing approaches can detect automatically detect mobile-friendly problems, but they are unable to repair the problem. It remains a manual effort to repair the mobile-friendly problems in a web page. To address this issue, Mahajan et al. propose an approach that first builds graph-based models of the layout of a web page. The constraints encoded by these graphs are used to find patches that can improve mobile friendliness while minimizing layout disruption. The approach leverages unique aspects of the problem domain to quantify metrics related to layout distortion and parallelize the computation of the solution to identify the best patch efficiently.

Different from the prior mentioned works, our work focuses on API-misuse repair, which belongs to another branch of domain-specific program repair.

6.3 Empirical Studies on Program Repair

In recent years, several studies have empirically evaluated the effectiveness of APR techniques [8, 9, 49]. Durieux et al. [9] conducted a large-scale experiment that evaluates 11 Java test-suite-based APR techniques on 2,141 bugs from 5 benchmarks. They found that these techniques can generate patches for a diverse number of bugs, and they are complementary to each other. Moreover, they found that the APR techniques perform significantly better on Defects4J than on the other benchmarks. Furthermore, they identified six primary reasons that these APR techniques fail to generate patches for bugs, including incorrect fault localization and multiple fault localization.

Motwani et al. [49] analyzed the effectiveness of APR techniques in real-world Java programs, especially on the defects made by the developers during their regular development process. Some of their findings are (1) APR techniques do sometimes produce patches, while those patches often break untested or undertested functionality; (2) The produced patches often overfit to the provided test suite. Their work outlines the shortcomings of existing APR techniques when applied in the real world.

Different from the empirical studies mentioned above, our work evaluates APR techniques in repairing a specific type of error, i.e., API misuse.

7 CONCLUSION AND FUTURE WORK

In this work, we present an empirical study that evaluates 11 learning-aided APR techniques on their capability to repair API misuse. We build a large-scale API misuse benchmark, which consists of two variants: the *complete data* with 118,490 pairs of the buggy and fixed methods, and the *single-line data*with 54,510 pairs of the single-line buggy and fixed methods. We conclude several findings based on the empirical results of existing approaches. We find that decoder-based and encode-decoder-based PLMs are more effective than encoder-based PLMs. Among all the 9 PLMs we investigate, CodeT5 achieves the highest scores in terms of EM. PLMs are more effective than the studied APR techniques.

We recommend that future work addresses the limitation of the current APR techniques by carefully considering handling long methods, improving bug localization, and prioritizing frequent and non-trivial API misuses. In the future, we are also interested in investigating alternative ways to adopt PLMs for repairing API misuse, such as prompt tuning [65]. By analyzing the results of the current approaches, we plan to propose new approaches to repair API misuse. Our dataset and the code are publically available at https://anonymous.4open.science/r/TOSEM-API-Misuse.

REFERENCES

[1] Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. 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* (2021), pp. 2655–2668.

[2] Amann, S., Nadi, S., Nguyen, H. A., Nguyen, T. N., and Mezini, M. Mubench: A benchmark for api-misuse detectors. In *Proceedings of the 13th international conference on mining software repositories* (2016), pp. 464–467.

[3] Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N., and Mezini, M. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering 45*, 12 (2018), 1170–1188.

[4] Bae, S., Cho, H., Lim, I., and Ryu, S. Safewapi: Web api misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering* (2014), pp. 507–517.

[5] Berabi, B., He, J., Raychev, V., and Vechev, M. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning* (2021), PMLR, pp. 780–791.

[6] Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering 47*, 9 (2019), 1943–1959.

[7] Clark, K., Luong, M.-T., Le, Q. V., and Manning, C. D. Electra: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations* (2019).

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

[9] Durieux, T., Madeiral, F., Martinez, M., and Abreu, R. Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 302–313.

[10] Elnaggar, A., Ding, W., Jones, L., Gibbs, T., Feher, T., Angerer, C., Severini, S., Matthes, F., and Rost, B. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443* (2021).

[11] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[12] Gao, Q., Zhang, H., Wang, J., Xiong, Y., Zhang, L., and Mei, H. Fixing recurring crash bugs via analyzing q&a sites (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015), IEEE, pp. 307–318.

[13] Gao, X., Noller, Y., and Roychoudhury, A. Program repair. *arXiv preprint arXiv:2211.12787* (2022).

[14] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 38–49.

[15] Gu, Z., Wu, J., Liu, J., Zhou, M., and Gu, M. An empirical study on api-misuse bugs in open-source c programs. In *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)* (2019), vol. 1, IEEE, pp. 11–20.

[16] Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., and Yin, J. 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)* (2022), pp. 7212–7225.

[17] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Shujie, L., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations* (2020).

[18] Haryono, S. A., Thung, F., Kang, H. J., Serrano, L., Muller, G., Lawall, J., Lo, D., and Jiang, L. Automatic android deprecated-api usage update by learning from single updated example. In *Proceedings of the 28th international conference on program comprehension* (2020), pp. 401–405.

[19] Herzig, K., Just, S., and Zeller, A. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *2013 35th international conference on software engineering (ICSE)* (2013), IEEE, pp. 392–401.

[20] Huang, Z., Lie, D., Tan, G., and Jaeger, T. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 539–554.

[21] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[22] Jiang, N., Lutellier, T., and Tan, L. Cure: Code-aware neural machine translation for automatic program repair. In

2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (2021), IEEE, pp. 1161–1173.

[23] Just, R., Jalali, D., and Ernst, M. D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (2014), pp. 437–440.

[24] Kang, H. J., and Lo, D. Active learning of discriminative subgraph patterns for api misuse detection. *IEEE Transactions on Software Engineering 48*, 8 (2021), 2761–2783.

[25] Kechagia, M., Devroey, X., Panichella, A., Gousios, G., and van Deursen, A. Effective and efficient api misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis* (2019), pp. 192–203.

[26] Kechagia, M., Mechtaev, S., Sarro, F., and Harman, M. Evaluating automatic program repair capabilities to repair api misuses. *IEEE Transactions on Software Engineering* (2021).

[27] Kenton, J. D. M.-W. C., and Toutanova, L. K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT* (2019), pp. 4171–4186.

[28] Le, X. B. D., Lo, D., and Le Goues, C. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)* (2016), vol. 1, IEEE, pp. 213–224.

[29] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering 38*, 1 (2011), 54–72.

[30] Lee, J., Hong, S., and Oh, H. Npex: repairing java null pointer exceptions without tests. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 1532–1544.

[31] Legunsen, O., Hassan, W. U., Xu, X., Roşu, G., and Marinov, D. How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2016), IEEE, pp. 602–613.

[32] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. 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* (2020), pp. 7871–7880.

[33] Li, G., Liu, H., Chen, X., Gunawi, H. S., and Lu, S. Dfix: automatically fixing timing bugs in distributed systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), pp. 994–1009.

[34] Li, X., Jiang, J., Benton, S., Xiong, Y., and Zhang, L. A large-scale study on api misuses in the wild. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)* (2021), IEEE, pp. 241–252.

[35] Li, Y., Wang, S., and Nguyen, T. N. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 511–523.

[36] Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 1035–1047.

[37] Li, Z., Machiry, A., Chen, B., Naik, M., Wang, K., and Song, L. Arbitrar: User-guided api misuse detection. In *2021 IEEE Symposium on Security and Privacy (SP)* (2021), IEEE, pp. 1400–1415.

[38] Lin, D., Koppel, J., Chen, A., and Solar-Lezama, A. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity* (2017), pp. 55–56.

[39] Liu, K., Koyuncu, A., Kim, D., and Bissyandé, T. F. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019), pp. 31–42.

[40] Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyandé, T. F., Kim, D., Wu, P., Klein, J., Mao, X., and Traon, Y. L. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (2020), pp. 615–627.

[41] Long, F., and Rinard, M. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2016), pp. 298–312.

[42] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C. B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual* (2021).

[43] Lutellier, T., Pham, H. V., Pang, L., Li, Y., Wei, M., and Tan, L. 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.

[44] Madeiral, F., Urli, S., Maia, M., and Monperrus, M. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), IEEE, pp. 468–478.

[45] Mahajan, S., Abolhassani, N., McMinn, P., and Halfond, W. G. Automated repair of mobile friendly problems in web pages. In *Proceedings of the 40th International Conference on Software Engineering* (2018), pp. 140–150.

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

[47] Monperrus, M. *The living review on automated program repair.* PhD thesis, HAL Archives Ouvertes, 2018.

[48] Monperrus, M., and Mezini, M. Detecting missing method calls as violations of the majority rule. *ACM Transactions on Software Engineering and Methodology (TOSEM) 22*, 1 (2013), 1–25.

[49] Motwani, M., Soto, M., Brun, Y., Just, R., and Le Goues, C. Quality of automated program repair on real-world defects. *IEEE Transactions on Software Engineering* (2020).

[50] Nguyen, H. A., Nguyen, T. T., Wilson Jr, G., Nguyen, A. T., Kim, M., and Nguyen, T. N. A graph-based approach to api usage adaptation. *ACM Sigplan Notices 45*, 10 (2010), 302–321.

[51] Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 772–781.

[52] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (2002), pp. 311–318.

[53] Pradel, M., and Gross, T. R. Leveraging test generation and specification mining for automated bug detection without false positives. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 288–298.

[54] Pradel, M., Jaspan, C., Aldrich, J., and Gross, T. R. Statically checking api protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 925–935.

[55] Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering* (2014), pp. 254–265.

[56] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog 1*, 8 (2019), 9.

[57] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P. J., et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res. 21*, 140 (2020), 1–67.

[58] Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[59] Saha, R. K., Lyu, Y., Lam, W., Yoshida, H., and Prasad, M. R. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories* (2018), pp. 10–13.

[60] See, A., Liu, P. J., and Manning, C. D. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2017), pp. 1073–1083.

[61] Tan, S. H., Dong, Z., Gao, X., and Roychoudhury, A. Repairing crashes in android apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), IEEE, pp. 187–198.

[62] Tan, S. H., and Roychoudhury, A. relifix: Automated repair of software regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), vol. 1, IEEE, pp. 471–482.

[63] Tufano, M., Watson, C., Bavota, G., Penta, M. D., White, M., and Poshyvanyk, D. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM) 28*, 4 (2019), 1–29.

[64] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems 30* (2017).

[65] Wang, C., Yang, Y., Gao, C., Peng, Y., Zhang, H., and Lyu, M. R. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 382–394.

[66] Wang, Y., Wang, W., Joty, S. R., and Hoi, S. C. H. 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, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021* (2021), Association for Computational Linguistics, pp. 8696–8708.

[67] Wen, M., Chen, J., Wu, R., Hao, D., and Cheung, S.-C. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th international conference on software engineering* (2018), pp. 1–11.

[68] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Online, Oct. 2020), Association for Computational Linguistics, pp. 38–45.

[69] Xia, C. S., and Zhang, L. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 959–971.

[70] Xiong, Y., Liu, X., Zeng, M., Zhang, L., and Huang, G. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th international conference on software engineering* (2018), pp. 789–799.

[71] Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (2022), pp. 1–10.

[72] Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S. L., Durieux, T., Le Berre, D., and Monperrus, M. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering 43*, 1 (2016), 34–55.

[73] Ye, H., Martinez, M., Luo, X., Zhang, T., and Monperrus, M. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2023), ASE '22, Association for Computing Machinery.

[74] Ye, H., Martinez, M., and Monperrus, M. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 1506–1518.

[75] Yuan, W., Zhang, Q., He, T., Fang, C., Hung, N. Q. V., Hao, X., and Yin, H. Circle: continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022), pp. 678–690.

[76] Yun, I., Min, C., Si, X., Jang, Y., Kim, T., and Naik, M. {APISan}: Sanitizing {API} usages through semantic {Cross-Checking}. In *25th USENIX Security Symposium (USENIX Security 16)* (2016), pp. 363–378.

[77] Zeng, Z., Tan, H., Zhang, H., Li, J., Zhang, Y., and Zhang, L. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022), pp. 39–51.

[78] Zhang, J., Panthaplackel, S., Nie, P., Li, J. J., and Gligoric, M. Coditt5: Pretraining for source code and natural language editing. In *37th IEEE/ACM International Conference on Automated Software Engineering* (2022), pp. 1–12.

[79] Zhang, Q., Fang, C., Ma, Y., Sun, W., and Chen, Z. A survey of learning-based automated program repair. *arXiv preprint arXiv:2301.03270* (2023).

[80] Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., and Kim, M. Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), IEEE, pp. 886–896.

[81] Zhu, Q., Sun, Z., Xiao, Y.-a., Zhang, W., Yuan, K., Xiong, Y., and Zhang, L. 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.