One Size Does Not Fit All: Multi-granularity Patch Generation for Better Automated Program Repair

Bo Lin*
linbo19@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Shangwen Wang*†
wangshangwen13@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Ming Wen†
mwenaa@hust.edu.cn
School of Cyber Science and Engineering
Huazhong University of Science and Technology
Wuhan, China

Liqian Chen*
lqchen@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Xiaoguang Mao*
xgmao@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Abstract

Automated program repair aims to automate bug correction and alleviate the burden of manual debugging, which plays a crucial role in software development and maintenance. Recent studies reveal that learning-based approaches have outperformed conventional APR techniques (e.g., search-based APR). Existing learning-based APR techniques mainly center on treating program repair either as a translation task or a cloze task. The former primarily emphasizes statement-level repair, while the latter concentrates on token-level repair, as per our observations. In practice, however, patches may manifest at various repair granularity, including statement, expression, or token levels. Consequently, merely generating patches from a single granularity would be ineffective to tackle real-world defects. Motivated by this observation, we propose `Mulpor`, a multi-granularity patch generation approach designed to address the diverse nature of real-world bugs. `Mulpor` comprises three components: statement-level, expression-level, and token-level generator, each is pre-trained to generate correct patches at its respective granularity. The approach involves generating candidate patches from various granularities, followed by a re-ranking process based on a heuristic to prioritize patches. Experimental results on the Defects4J dataset demonstrate that `Mulpor` correctly repair 92 bugs on Defects4J-v1.2, which achieves 27.0% (20 bugs) and 12.2% (10 bugs) improvement over the previous state-of-the-art NMT-style Rap-Gen and Cloze-style GAMMA. We also investigated the generalizability of `Mulpor` in repairing vulnerabilities, revealing a notable 51% increase in the number of correctly-fixed patches compared with state-of-the-art vulnerability repair approaches. This paper underscores the importance of considering multiple granularities in program repair techniques for a comprehensive strategy to address the diverse nature of real-world software defects. `Mulpor`, as proposed herein, exhibits promising results in achieving effective and diverse bug fixes across various program repair scenarios.

CCS Concepts

• **Software and its engineering → Software maintenance tools**; *Software defect analysis*; Software testing and debugging.

Keywords

Automated Program Repair, Pre-Training, Deep Learning.

ACM Reference Format:
Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. 2024. One Size Does Not Fit All: Multi-granularity Patch Generation for Better Automated Program Repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680381

1 Introduction

Modern software systems are witnessing an unprecedented surge in complexity and scale, a phenomenon underscored by the escalating prevalence of software bugs [23, 43]. These bugs not only impede the progress of software development but also yield a myriad of challenges, ranging from user dissatisfaction to financial losses [17]. The manual identification and fixing of these bugs demand substantial investments of time and resources [2]. In response to this challenge, Automated Program Repair (APR) has emerged as a pivotal field in software engineering, aiming to automate the correction of identified buggy code snippets and thereby alleviate the arduous burden of manual debugging activities [17].

*Bo Lin, Shangwen Wang, Liqian Chen and Xiaoguang Mao are also with the State Key Laboratory of Complex & Critical Software Environment
†Shangwen Wang and Ming Wen are the corresponding authors.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680381

Over recent decades, traditional APR techniques have shown promising results. They have evolved into various types, including search-based [26, 38, 45, 45, 54] and constraint-based techniques [39, 41, 44, 59]. Recently, the adoption of advanced deep learning techniques for program repair has emerged as a notable trend. Particularly, a number of studies have shown that leveraging the power of deep learning can be more proficient at fixing bugs than traditional template-based approaches [58, 64]. In the literature, there are mainly two ways applying deep learning models on the program repair task. On the one hand, several techniques treat program repair as a Neural Machine Translation (NMT) task to transform the buggy program into the fixed version, with an emphasis on translating the whole buggy **statement** into a correct one [22, 40, 49, 61]. For instance, RewardRepair [61] uses a loss function based on compilation and execution information to construct an NMT model, which takes a suspicious statement as the input and outputs the fixed statement. Consequently, such techniques generate patches at the statement level. On the other hand, some latest studies utilize the weapon of LLMs and treat program repair as a cloze-style task, in which the buggy **token** is replaced by the predictions from LLMs based on the context around [58, 63]. Such techniques achieve relatively high performances when there is only one buggy token in the original program, probably because during the pre-training, the masked language modeling task requires the model to predict the content for a single masked token. To explore this point, we manually analyzed the correct patches generated by GAMMA [63], a state-of-the-art APR technique, and found that 76% of such patches focused on token-level change from the original buggy program. In this regard, such learning-based techniques could be considered as generating patches at the token level.

Despite being effective, existing techniques ignore the fundamental property of bug-fixing activities, i.e., different bugs are required to be fixed at different granularities [48]. Specifically, sometimes the bug is required to be fixed at the statement level. In other cases, simply modifying a code token in the faulty statement can correct the behavior of the program. For the former cases, correct patches would not be generated by approaches that produce patches at the token level, since these approaches operate at a finer-grained granularity (token vs. statement), and the search space of which will not contain the correct patch. In contrast, for the latter cases, completely rewriting the statement would reduce the probability of generating correct patches. This is primarily due to the significantly larger search space these approaches have to navigate (statement v.s. token), which makes it challenging for them to accurately identify the correct repair actions. To gain a comprehensive understanding towards such a multi-granularity property, we performed an exploratory experiment (discussed in Section 3) on a dataset containing 787,178 bug-fixing instances collected from open-source projects [49]. Results reveal that bugs from real-world are often required to be repaired at various granularities, including the statement, expression, and token level. Moreover, patches generated at different granularities share similar percentages in the dataset, suggesting that no specific granularity dominates over the others. These findings suggest that *merely relying on the ability to produce patches at a specific granularity may not be sufficient to effectively address real-world software defects*. Therefore, it is crucial to consider multiple granularities when devising program repair techniques.

Motivated by our observations, in this paper, we propose `Mulpor`, a Multi-granularity patch generator. It is simple yet effective that can produce patches from various granularities, aiming at effectively handling the diverse nature of real-world bugs. Our key idea is to generate patches at various granularities, increasing the likelihood of producing the correct patches, and then re-rank the candidate patches based on heuristics to ensure the correct patch is recommended at the top positions. We foresee that an ideal resolution would be predicting the repair granularity first, followed by selecting the corresponding generator. Nonetheless, predicting the repair granularity of a buggy program is non-trivial. For instance, due to this challenge, existing template-based APR techniques [18, 30, 37] opt to traverse all the fix patterns at various repair granularities when performing patch generation. To achieve its target, `Mulpor` integrates three components, i.e., the statement-level, expression-level, and token-level generator. Each component is pre-trained to fill in masked contents at its respective granularity, thus enabling them to generate correct programs across different granularities. When provided with a buggy statement, `Mulpor` employs the three components individually. In particular, it invokes the statement-level generator to regenerate the entire statement, the expression-level generator to modify the expressions within the statement, and the token-level generator to update the code tokens in the statement. This process generates a set of candidate patches, followed by a patch re-ranking process where the candidate patches are prioritized according to their similarity to the original buggy program. The rationale of this step is that existing studies have shown that correct patches often share certain similarities with the original buggy program and do not require significant changes [31, 50]. We thus prioritize patches that show the highest degree of similarity to the original program. To obtain the final results, we check the ability of each candidate patch to pass the test suite, and then manually examine all plausible patches to confirm that the bugs are correctly fixed.

We evaluate the effectiveness of `Mulpor` on the widely-adopted Defects4J-v1.2 benchmark. The results show that `Mulpor` outperforms all studied APR approaches, repairing 10 more bugs (82 → 92) with perfect FL and 6 more bugs (50 → 56) without perfect FL than the previous state-of-the-art models. Furthermore, we conducted an in-depth exploration of `Mulpor`'s capabilities across a broader spectrum of code repair scenarios, specifically focusing on vulnerability repair. The findings reveal that `Mulpor` produces a notable 51% more correct patches compared to the existing state-of-the-art vulnerability repair approach VulRepair+ [24]. In summary, our study makes the following contributions:

- **New Dimension:** Our work reveals the multi-granularity nature of bug-fixing activities, and emphasizes that APR tools should consider the design of various granularities, such as statement, expression, and token levels.
- **State-of-the-art APR tool:** We propose a multi-granularity patch generation approach, `Mulpor`. `Mulpor` initially acquires semantic and syntactic knowledge through intricate unsupervised pre-training tasks at different granularities. Subsequently, it undergoes fine-tuning on real-world bug-fixing datasets. This enables `Mulpor` to generate patches at various granularities, thereby increasing the likelihood of producing correct patches.

- **Extensive Study:** `Mulpor` is extensively evaluated on Defects4J, showcasing superior bug repair performance. The study extends to vulnerability repair, revealing significant advancements over current state-of-the-art approaches. The results affirm the effectivenss of `Mulpor` across diverse program repair challenges.

2 Background and Related Works

2.1 Automated Program Repair

Since the inception of Genprog [33], a wide array of APR techniques have been proposed, contributing to the reduction of manual debugging efforts and the automatic generation of patches. According to the study from Goues *et al.* [19], APR techniques can be divided into three categories, namely search-based repair [26, 33, 38, 45, 54], constraint-based repair [39, 41, 44, 59], and learning-based repair [3, 8, 27, 34, 40, 64]. Search-based repair methods stand out as conventional APR techniques, often relying on program modification through mutation with heuristic algorithms [45] or genetic programming [33] to generate a multitude of candidate fixes for validation via unit tests. To narrow down the search space in search-based repair, search strategies have involved to incorporate fix patterns mined using redundancy-based techniques [26, 32, 38, 39, 55] from history data, such as commits [32], existing codebase [26] and even external Q&As from StackOverflow [38]. In contrast to heuristic repair techniques, constraint-based methods proceed by formulating repair constraints that the patched code should satisfy. Constraint solving or other search techniques are then employed to identify solutions to these repair constraints.

In recent years, the field of APR has witnessed significant advancements, primarily driven by the proliferation of deep learning techniques. Most learning-based APR methods [3, 8, 27, 34, 40, 64] aim to streamline the program repair process by framing it as a NMT task, which typically translates a faulty statement into a correct one. These NMT-style APR techniques use an encoder-decoder architecture [10]. The encoder extracts representations from the buggy code, and the decoder generates the corrected code based on these representations. As an illustration, Rap-Gen [51] adopts the CodeT5 model along with a similar patch search strategy, achieving state-of-the-art performance in NMT-style APR techniques. The effectiveness of these techniques owes much to the remarkable capacity from deep learning techniques to discern complex relationships within vast code corpora. Hence, these techniques have consistently achieved state-of-the-art performance in recent years.

However, despite the promising performance, NMT-style APR techniques still face certain limitations, primarily stemming from the quality and quantity of available historical bug-fixing pairs used for training [58]. Recognizing this, Xia *et al.* recently introduced AlphaRepair [58], which formulates program repair as a cloze task. This approach involves directly predicting the correct code based on contextual information, using pre-trained models. While AlphaRepair capitalizes on existing pre-trained models, which were not specifically designed for program repair, they are ill-suited for handling complex code repair. For instance, CodeBERT [15], as used in AlphaRepair, only supports filling in one token at a time. Within this context, these techniques can be considered as generating patches at the token level. However, in practice, program repair frequently requires changes at multi-granularity, such as modifying statements or expressions [37]. In this context, we present `Mulpor`, a simple yet effective learning-based APR technique. It augments the model with the capability to generate patches at various levels of granularity, therefore increasing the likelihood of producing the correct patch compared to generating patches at a fixed granularity.

2.2 Pre-Training Techniques

Training deep learning models from scratch often requires extensive labelled data, which can be resource-intensive and time-consuming. In response, the self-supervised pre-training techniques has emerged as a powerful alternative. Self-supervised objectives are designed to endow models with common-sense knowledge by leveraging large amount of unlabelled data. These pre-trained models can subsequently be fine-tuned on specific downstream tasks with a relative modest amount of labelled data. Initially pioneered in Natural Language Processing [11, 46], self-supervised pre-training has evolved to address code-related tasks, with a number of pre-training tasks that are tailored to instill models with domain-specific programming language knowledge being introduced [15, 21, 52]. One of the pioneering self-supervised objectives, the Masked Language Modeling (MLM) [11], gained considerable popularity within the research community. In MLM, a portion of the training data is masked, and the primary training objective resolves around predicting or recovering the original data. MLM has showcased its effectiveness in tasks related to language [11, 20, 21].

A promising avenue in pre-training technologies involves the application of self-supervised techniques to program repair. Researchers have proposed leveraging pre-trained code models to formulate program repair as a cloze task. This involves masking a buggy code snippet and prompting the model to predict the correct code snippet within the given context. For instance, AlphaRepair [58] utilizes CodeBERT [15], and GAMMA [63] leverages UnixCoder [20] for donor code retrieval through a mask prediction task without additional pre-training. However, pre-trained models used in cloze-style APR have inherent limitations from their pre-training objectives. They primarily focus on restoring one or several tokens from masked locations. Operating at the individual token level restricts their capacity to grasp broader contextual information, including relationships between groups of words [28]. This limitation extends to their ability to address more complex bugs that require to rewrite a complete statement. Consequently, these cloze-style APR techniques are generally more effective for token-level repairs. To better understand the limitation of cloze-style APR, we also perform an investigation on the patches generated by GAMMA for the defects from Defects4J-v1.2 benchmark. The results indicate that approximately 67% of the patches and 76% of the correct patches generated by GAMMA focus on token-level modifications, demonstrating its inability to perform complex repairs. However, it is crucial to recognize that the repair granularity of buggy programs can vary [37, 57], encompassing levels such as statement and expression granularity. This variability implies that pre-trained models based on MLM may encounter limitations in addressing program repair tasks at a coarse-grained level, such as the statement level. This observation underscores the necessity for specialized pre-training models explicitly designed to accommodate the diverse repair granularities encountered in program repair tasks.

3 MOTIVATING EXAMPLES

In existing literature, there are two primary ways to apply deep learning models in the field of program repair. The first is to treat program repair as an end-to-end NMT task, aiming to transform a buggy program into its corrected version [27, 40, 58]. The second views program repair as a cloze-style task, wherein the faulty token is replaced by predictions from LLMs based on the contextual information surrounding it. To elucidate the limitations of current learning-based APR techniques, we present several illustrative cases, each operating at different repair granularities. We also select the patches generated by state-of-the-art techniques in the cloze-style APR (i.e., GAMMA [63]) and NMT-style APR (i.e., Rap-Gen [51]) to illustrate the limitations of existing approaches.

Repair granularity denotes the kinds of code entities that are directly modified by different repair actions [37]. Following the templates derived from historical bug-fixing by experts [37], the repair granularity can be categorized into three types: statement-level, expression-level, and token-level. Statement-level repair refers to the alteration of the statement-level node by actions such as addition, deletion, or modification. Expression and token-level repairs mirror similar changes but at the respective node levels. Note that token-level repairs may change more than one token in the buggy program. Such as Chart-10 bug [1] from Defects4J, where an identifier is updated to a method invocation. But this repair can be mapped into the modification of leaf nodes in the buggy Abstract Syntax Tree (AST), i.e., token-level repair.

Figure 1 illustrates real bugs from the widely-used Defects4J benchmark [29] at different repair granularities. For instance, in addressing the Chart-12 bug, the developer replaces an old statement with a new one, directly modifying the `ExpressionStatement`, indicating a statement-level repair. However, when GAMMA utilizes CodeBERT as the base model, it struggles to generate a correct patch due to the limited search space. Since GAMMA only masks the identifier `dataset` guided by a fix pattern, its search space only involves updating this identifier with other candidates, which is inadequate for repairing this bug. Similarly, in case of the Closure-123, repairs are associated with an expression node. The developer updates a `FieldAccess` expression into a `MethodInvocation` expression. GAMMA tends to select a fix pattern that replaces the identifier `OTHER` rather than the entire `FieldAccess` expression in the buggy statement. In contrast, RAP-Gen aims to translate a buggy statement into a fixed one, which leads to a complete rewrite of the statement and thus introduces unnecessary changes. Token-level repair in the Chart-24 bug exemplifies the limitation of Rap-Gen further, where the correct patch updates the identifier `value` to `v`. However, Rap-Gen rewrites the entire buggy statement, setting the `value` to the minimum value between `v` and `this.lowerBound`.

In stark contrast to existing approaches that mainly focus on a single repair granularity, various repair granularities exist in the real world. In order to better understand this phenomenon, we first perform an investigation into the repair distribution across different granularities in historical data. This investigation was conducted using GumTree [13] and computed on 787,178 bug-fixing pairs obtained from open-source projects provided by Tufano *et al.* [49]. Table 1 presents the proportion of repairs at different granularities,

[1] https://program-repair.org/defects4j-dissection/#!/bug/Chart/10/

Statement-level repair from Chart-12:

```
// Oracle Patch
- this.dataset = dataset;
+ setDataset(dataset);
// Patch generated by GAMMA
- this.dataset = dataset;
+ this.data = dataset;
```

Expression-level repair from Closure-123:

```
// Oracle Patch
- Context rhsContext = Context.OTHER;
+ Context rhsContext = getContextForNoInOperator(context);
// Patch generated by GAMMA
- Context rhsContext = Context.OTHER;
+ Context rhsContext = Context.STATEMENT;
// Patch generated by Rap-Gen
- Context rhsContext = Context.OTHER;
+ ContextFactory rhsContext = getContext(context, false);
```

Token-level repair from Chart-24:

```
// Oracle Patch
- int g = (int) (( value  - this.lowerBound) /
      (this.upperBound - this.lowerBound) * 255.0);
+ int g = (int) (( v  - this.lowerBound) /
      (this.upperBound - this.lowerBound) * 255.0);
// Patch generated by Rap-Gen
- int g = (int) ((value - this.lowerBound) /
      (this.upperBound - this.lowerBound) * 255.0);
+ value = Math.min(v, this.lowerBound);
```

Figure 1: Program repair at different granularities.

along with the top three repair types in each category. Notably, statement, expression, and token-level repairs constitute 30.8%, 33.6%, and 35.6% of the repairs, respectively. This distribution reveals that existing approaches primarily operating at a single granularity level (e.g., token-level) may limit their ability to support the patch generation at different granularities, thereby impacting their performance in patch generation. Based on these observations, we hypothesize that the effectiveness of APR approaches could be enhanced by generating patches at different granularities, aiming at effectively handling the diverse nature of real-world bugs. Our key idea is to first generate patches from various granularities, which increases the probability of generating the correct patch compared to a fixed granularity. We then re-rank these candidate patches from various granularities based on heuristics aims to maximize the likelihood of recommending the correct patch at the top.

4 Mulpor

In this section, we present a detailed overview of `Mulpor`, the workflow of which is depicted in Figure 2. During the pre-training phase, we initially extract all functions provided by the CodeSearchNet [25] dataset, encompassing more than 2 million multi-linguistic functions. Given that `Mulpor` comprises three integral components, i.e., the statement-level generator, expression-level generator, and token-level generator, all the functions are used as the pre-training data at three distinct granularities, which facilitates the training of the corresponding generators (Section 4.2). In the pre-training phase, each component is pre-trained to fill in masked contents at its respective granularity. This process enables the components to accurately generate programs across diverse granularities.

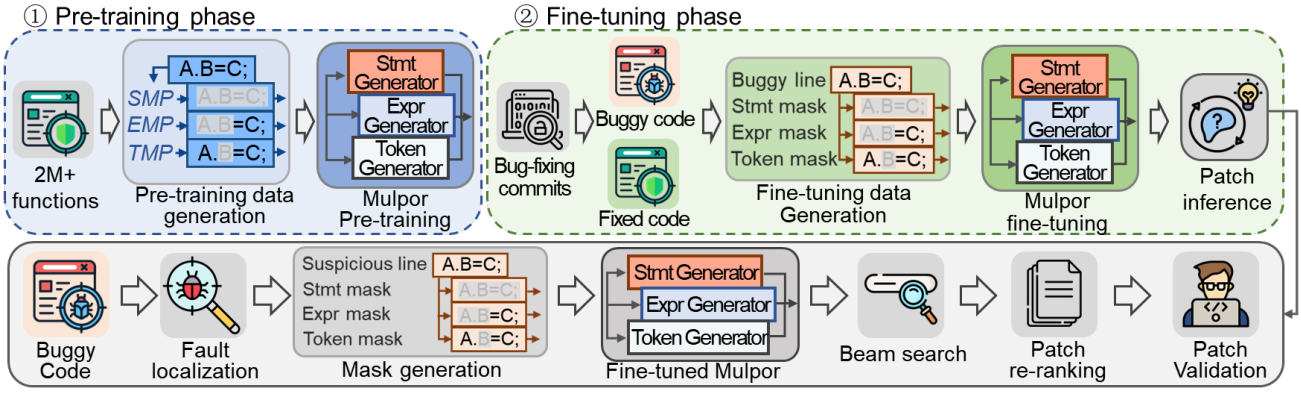

Figure 2: The overall workflow of `Mulpor`.

Table 1: The proportion of repairs at different granularities.

Granularity	Type†	Proportion(in %)
Statement	Expression Statement	17.3
	If Statement	7.8
	Return Statement	3.3
	Others	2.4
	Total	30.8
Expression	Method Invocation	20.1
	Variable Declaration	5.7
	Assignment	3.1
	Others	4.7
	Total	35.8
Token	Identifier	19.4
	Type Literal	5.7
	Modifiers	5.7
	Others	4.8
	Total	33.4

† Types are from Eclipse JDT AST parser.

In the fine-tuning phase, we reuse the BFP [49] and VulRD [16] dataset for bug repair and vulnerability repair, respectively. These datasets contain both buggy and fixed code from real-world bug-fixing commits. Subsequently, we generate data to fine-tune `Mulpor` (Section 4.3). During the patch generation process, a buggy program and a set of test suites, inducing failures in the program, are utilized to yield a list of suspicious code lines through fault localization approaches. We then use the fine-tuned `Mulpor` to employ three generators and a beam search strategy to generate candidate patches for these suspicious elements (Section 4.4). Afterwords, we re-rank the candidate patches utilizing a heuristic to ensure that the correct patch is prioritized at the top positions (Section 4.5). Finally, `Mulpor` employs various strategies to validate the correctness of candidate patches across different scenarios (Section 4.6).

4.1 Model Architecture

`Mulpor` consists of three generators at statement, expression and token level. Following the T5 model [47], each generator uses an encoder-decoder architecture. Both the encoder and decoder have 12 Transformer layers. In each layer, 12 attention heads are utilized for the multi-head attention computation, resulting in a total parameter size of 220M. Such an architecture is widely-used by state-of-the-art pre-trained models [6, 16, 36, 52]. As per existing studies [16, 36, 51], we initialize the parameters of each generator with the weights from CodeT5, with the aim to equip the model with certain domain knowledge of programming languages.

4.2 Pre-Training Tasks

While existing pre-training techniques either focus on masked token prediction [15, 21] or masked span prediction [28, 47, 52] (typically less than 6 tokens), our study on the BFP dataset [49] shows that bug fixing in real-world scenarios can happen at various granularities. To effectively handle the diverse nature of real-world bugs, our key idea is to generate patches from various granularities, thereby increasing the likelihood of producing the correct patch compared to generating patches from a fixed granularity. To that end, we design three pre-training tasks at the statement-level, expression-level, and token-level to equip our approach with the capacity to predict patches at the corresponding granularity. Note that we opted to mask 15% of the elements in different granularities because it is a common practice adopted by previous studies [11, 15, 21, 52]. According to the study [56], this masking rate provides sufficient context to learn satisfactory representations.

4.2.1 Statement-level mask prediction (SMP). In this pre-training task, we delve into the realm of statement-level mask prediction, a refinement of the traditional masked span prediction. Unlike its predecessor, where code tokens or code spans were concealed (usually less than 6 tokens [47, 52]), statement-level mask prediction operates at a coarser-grained level by obfuscating individual code statements. To perform this pre-training task, we begin with the input code that is augmented with statement-level masks. These masks are strategically placed to target code statement rather than randomly selected code span without integrity syntactic structure. Specifically, to generate these masks, we utilize a well-regarded AST parser, tree-sitter[2], to extract all statements from the input code. Subsequently, a randomly selection process is implemented, where 15% of these statements are masked, rendering them temporarily invisible to the model. The primary objective of this task is to equip the model with a broader understanding of token distribution within masked statements while maintaining the contextual coherence. Formally, the loss can be described as:

$$\mathcal{L}_{SMP}(\theta) = \sum_{i=1}^{k} -logP_\theta(s_i|s^{mask}, s_{<i})$$

[2] https://tree-sitter.github.io/tree-sitter/

where s^{mask} is the masked statement, k denotes the number of code tokens in masked statement, and $s_{<i}$ is the token sequence predicted for the masked statement so far.

4.2.2 Expression-level mask prediction (EMP). In this pre-training task, we extend our exploration to a finer granularity by introducing EMP. This task builds upon the SMP by further refining the masking process at the level of individual code expressions. Similar to SMP, we leverage the tree-sitter parser to extract all expressions from the input code. However, unlike SMP, EMP operates at the expressions within statements and focuses on understanding and predicting the masked expressions. Approximately 15% of these expressions are randomly selected and masked, challenging the model to generate the correct expression based on the contextual information provided. The objective of EMP is to enhance the model's ability to grasp the domain knowledge of expression-level syntax and semantics. This finer granularity is expected to contribute to the proficiency of model in generating accurate expression in program repair. Formally, the loss can be described as:

$$\mathcal{L}_{EMP}(\theta) = \sum_{i=1}^{k} -logP_\theta(e_i|e^{mask}, e_{<i})$$

where e^{mask} is the masked expression, k denotes the number of code tokens in masked expression, and $e_{<i}$ is the token sequence predicted for the masked expression so far.

4.2.3 Token-level mask prediction (TMP). To generate patches at a finer granularity, we introduce TMP as a pre-training task. While SMP and EMP are at higher levels of abstraction, TMP focuses on individual code tokens. This task aims to refine the model's understanding of token-level syntax and semantics. To implement TMP, we employ the tree-sitter to extract all tokens from the input code. Subsequently, we randomly mask 15% of these tokens following an existing study [15, 52]. This challenges the model to predict the correct token in the context of code, fostering a more granular understanding of code. By honing its ability to discern token-level relationships, the model is better equipped to generate accurate code during program repair. Formally, the loss can be described as:

$$\mathcal{L}_{TMP}(\theta) = \sum_{i=1}^{k} -logP_\theta(t_i|t^{mask}, t_{<i})$$

where t^{mask} is the masked input, k denotes the number of code tokens in masked input, and $t_{<i}$ is the token sequence predicted for the masked input so far.

4.3 Fine-Tuning

While the pre-training phase equips the model with the ability to correct faulty code based on its context, it is noteworthy that the pre-training dataset is not collected from real-world bug-fixing commits. Also, the masked locations in pre-training tasks are chosen randomly, not reflecting real-world fault locations. To bridge this gap, we fine-tune the model on the dedicated dataset constructed from real-world bug-fixing commits. This strategy aims to align with the mask granularities employed in pre-training tasks, also guiding the model to focus more on error-prone code snippets. For instance, at the statement level, our pipeline mirrors the SMP task, with a key distinction: instead of randomly selecting masked statement, we exclusively mask the statements affected during the code repair process. To achieve this, we employ GumTree [13] to compare the buggy and corrected code, identifying the range of code changes, and subsequently masking all statements within this identified range. Thus `Mulpor` learns how to predict and fill in the masked statements. This approach is extendable to expression and token levels, where masking is applied to the expressions and tokens within the identified range of code modifications.

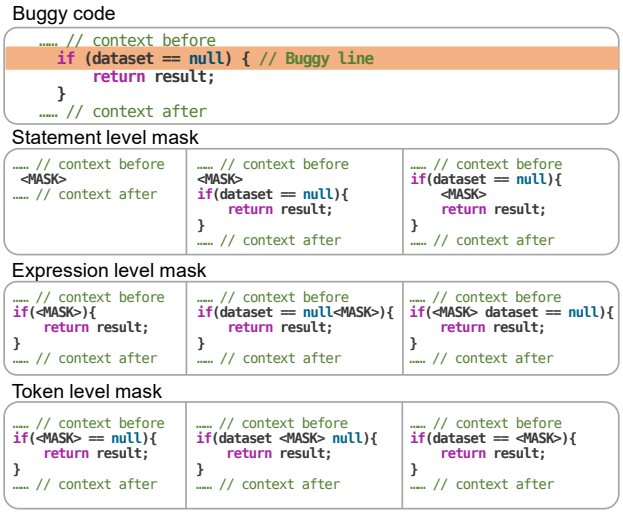

Figure 3: Mask generation at different granularities.

4.4 Patch Generation

Given a buggy code program, fault localization techniques will first return a list of suspicious code element, we then mask the faulty line at three granularities with strategies. Considering that fault localization is usually developed as an independent field and existing APR techniques employ off-the-shelf fault localization tools in the repair pipeline, we do not discuss the fault localization below and reuse the GZoltar API [5] and Ochiai similarity coefficient [1], which are commonly used in existing APR tools [26, 35, 55, 59, 64]. For perfect localization setting, we provide the location of buggy lines directly following the previous studies [8, 37, 58, 63]. Figure 3 shows a mask generation example for Chart-1 bug in Defects4J at three granularities: statement, expression and token-level.

Statement-level. At the statement level, we employ three strategies to generate the mask. The first is replacing the whole buggy statement. We refer to this as statement replacement since we query the `Mulpor` to generate a new statement directly to replace the buggy statement. The rest two strategies is to generate mask statement where we add `<MASK>` before/after the buggy line. These represent bug fixes where a new statement is inserted before/after the buggy location.

Expression-level. At the expression level, we employ the strategy of replacing all expressions in the faulty line with `<MASK>` for model querying, a process referred to as expression update. Additionally, in the generation of the expression-level mask, we account for insertions both before and after the erroneous expression by introducing `<MASK>` at the corresponding positions. It is noteworthy

that expression insertion is inherently synonymous with expression update. However, we emphasize the explicit specification of the insertion location, as it provides the model with more guidance (e.g., the insert location) and additional information (e.g., the old expression). This approach is intended to enhance the likelihood of generating the correct expressions.

Token-level. At the token level, we employ the strategy to restore the correct token from a masked location. This strategy involves replacing each token in the buggy line with `<MASK>`. The objective is to address token-level repairs, wherein modifications can be seamlessly mapped to the alteration of leaf nodes in the AST.

During inference, for each input generated by the aforementioned mask generation process, we employ beam search to generate a ranked list of candidate patches, where the number of predictions is determined by the beam size $\mathcal{B}$ (the setting of $\mathcal{B}$ is illustrated in 4.7). At each decoding step, the beam search algorithm selects the most $\mathcal{B}$ promising candidate patches with the highest probability using a *best-first search strategy* until the terminator which notifying the end of sentence is emitted. For each input generated by the corresponding strategies at three granularities, we generate $\mathcal{B}$ patches, which are subsequently re-ranked and merged (as detailed in Section 4.5) to form a final ranked list of candidate patches.

4.5 Patch Re-ranking

In the realm of APR, selecting and prioritizing candidate patches is paramount to enhancing the efficiency of software maintenance. In this pursuit, we introduce a patch ranking process to consolidate the outcomes generated by three different generators and prioritizing the correct patch at the top of the list whenever possible. We are inspired by prior research [31, 50], which establishes that more effective repairs often entail minimal code modifications. Additionally, insights from previous learning-based APR approaches [34, 35, 42] suggest a higher likelihood of correctness for patches positioned closer to the front of the generated patch list. Based on such observations, our approach incorporates a mixed ranking strategy to guide patch re-ranking. Specifically, each patch p is assigned a score tuple $S_p = (R_p, D_p)$, where R_p denotes the rank of patch within its respective generator, and D_p represents the edit distance between the original source code and the resulting patched code. The final patch list is sorted in ascending order based on the scores of the patches. In cases where multiple patches share the same R_p, a secondary sorting is performed based on the second element D_p.

4.6 Patch Validation

We validate patches by applying the corresponding changes to the buggy code for each generated candidate patch. We then compile each patched buggy code and filter out those that fail to compile. We then run the test cases against each compiled patched buggy code, seeking plausible patches that pass all the tests. We manually examine all plausible patches to ensure the bugs are fixed correctly, confirming that the patches are semantically equivalent to developer patches.

4.7 Implementation Details

Our model is implemented with PyTorch framework[3]. All the experiments are performed on a server with 2 NVIDIA GeForce RTX 4090 GPUs. The learning rate and batch size in the pre-training stage are set to 5e-5 and 32, respectively. When fine-tuning our `Mulpor` on the downstream tasks, we use a batch size of 32 and a learning rate of 2e-5, which are the same as we used during the pre-training stage. Regarding the timeouts used for the repair, the learning-based APR approaches [34, 58, 63, 64] typically set the running-time limit to five hours for fixing each bug, while TBar [37] is set to three hours. For a fair comparison, we set the running-time limit to the lowest value used in the baselines, which is three hours. In the patch inference phase, we configure the beam size to 15 to narrow the search space at each granularity (discussed in Section 7.2). This beam size is smaller than the values of 250 used in GAMMA [63] and 1000 used in CURE [27] and CoCoNuT [40]. Note that unlike existing learning-based approaches that predominantly focus on a single repair granularity, `Mulpor` generates patches across multiple granularities. Consequently, `Mulpor` faces the challenge of navigating a more extensive search space during patch generation, potentially resulting in the generation of more invalid patches. To mitigate this threat, we reduce the search space at each granularity by decreasing the beam size during patch generation. Specifically, while GAMMA generates an average of 475.8 patches for each bug in the Defects4J-v1.2 dataset, our approach generates an average of 87.6 patches. This observation indicates that, although the overall search space of `Mulpor` is larger than that of the tools operating at a fixed granularity, `Mulpor` can more effectively locate correct patches within its search space.

5 Study Design

5.1 Research Questions

In this paper, we seek to answer the following research questions:

RQ1: Compare with state-of-the-art APR approaches. How does `Mulpor` perform in repairing general bugs in open source projects compared with other APR approaches?

RQ2: Generalizability of `Mulpor`. How is the generalizability of `Mulpor` in repairing software vulnerability?

RQ3: Ablation study. What are the contributions of the major components of `Mulpor`?

5.2 Settings

5.2.1 Bug Repair. To answer RQ1, which evaluates the performance of `Mulpor` in general bug repair compared to other APR approaches, we report the baselines and evaluation metrics.

Baselines. In our comparative analysis, we assess the performance of `Mulpor` compared to both traditional and learning-based APR techniques. We choose nine recent Learning-based models including RAP-Gen [51], Repilot [53], AlphaRepair [58], Recoder [64], CURE [27], CoCoNuT [40], RewardRepair [61], SelfAPR [60], and GAMMA [63]. Additionally, to represent the traditional APR landscape, we incorporate two state-of-the-art template-based APR tools, namely TBar [37] and PraPR [18].

[3] https://pytorch.org/

Metrics. We compute how many bugs can be correctly fixed on Defects4J based on unit testing and manually verification which means the generated patch is semantically or syntactically equivalent to the developer patch by following the standard practice in APR research. Specifically, we first run test suites to automatically identify possible patches for each bug, and then manual checking to completely verify its correctness. The all predictions of `Mulpor` are included in our artifacts. We reuse the released results of baselines from the most recent work [51, 63] instead directly running the APR tools following the common practice in the APR community [27, 40, 62, 63].

5.2.2 Vulnerability repair. To answer RQ2, which evaluates the generalizability of `Mulpor` in software vulnerability repair compared to other vulnerability repair approaches, we report the selected baselines and evaluation metrics.

Baselines. As baselines of our comparative analysis, we choose four state-of-the-art learning-based vulnerability repair tools: VRepair [7], SeqTrans [9], VulRepair [16] and VulRepair+ [24]. VRepair and SeqTrans are deep learning-based approaches that tackle insufficient training data issues in software vulnerability repair through transfer learning. They undergo pre-training on a bug repair dataset and subsequent fine-tuning on a vulnerability repair dataset. While VRepair takes the code sequence as input, SeqTrans utilizes def-use chains to construct code sequences, capturing syntax and structure information around vulnerabilities with fewer noises. VulRepair, on the other hand, is a T5-based automated software vulnerability repair approach that employs a pre-trained model to address insufficient training data. It also utilizes the BPE algorithm to mitigate the Out-Of-Vocabulary problem. Huang et al. [24] conducted an empirical study, proposing modifications to the output format of VulRepair and introducing the ensemble strategy by combining multiple checkpoints. We have incorporated these enhancements into our extended version and named it VulRepair+.

Metrics. In vulnerability repair, due to the lack of test cases in the dataset, we evaluate the vulnerability repair performance of the VRepair [7] and VulRepair [16] approaches using Perfect Prediction (PP) accuracy following the previous studies [7, 9, 16]. PP measures the percentage of vulnerable functions for which an approach can generate vulnerability repairs that exactly match the ground-truth data (i.e., syntactic equivalence), which is human-written repairs in the VulRD dataset. Additionally, evaluating repair correctness is a highly time-consuming process. Therefore, we aim to improve the top-ranked repairs as much as possible. To measure the Perfect Prediction accuracy at different prediction levels, we utilize PP@Top1, PP@Top-5, and PP@Top-10. These metrics represent the PP accuracy at the Top-1, Top-5, and Top-10 predictions respectively. For all metrics, we present the results on a scale of 0-100 (%), where a higher score indicates better performance.

5.3 Dataset

To assess the effectiveness of `Mulpor`, we first pre-train it using multi-granularity pre-training tasks on CodeSearchNet [25] dataset, and then fine-tune it on the training set of the BFP [49] dataset. Defects4J is used to measure the effectiveness `Mulpor` in bug repair. To evaluate it performance in vulnerability repair, we use the VulRD dataset from VulRepair [16] for fine-tuning. We report their data statistics in Table 2.

Table 2: Statistics of the datasets.

Dataset	Train	Valid	Test
CodeSearchNet	454,451	15,328	-
BFP	58,909	6,546	-
Defects4J-v1.2	-	-	395
Defects4J-v2.0	-	-	430
VulRD	4,206	600	1,202

5.3.1 CodeSearchNet. CodeSearchNet [25] serves as a widely acknowledged pre-training dataset for code-related models [15, 52] and stands as a pivotal resource in the realm of deep learning for programming tasks. Comprising over 2 million data pairs from six languages (Ruby, JavaScript, Go, Python, Java, and PHP), the dataset is distributed in 80-10-10 proportions for training, validation, and testing. In our study, we exclusively utilize the training set, meticulously excluding instances used in fine-tuning to prevent data leakage and ensure the robustness of `Mulpor` training.

5.3.2 BFP. BFP [49] is a dataset for method-level bug repair, extracted from GitHub commits from 2011-2017 using GitHub Archive and GumTree [13]. It contains 65,455 validated bug-fixes, split in a 90-10 ratio for training and validation as shown in Table 2.

5.3.3 Defects4J. Defects4J [29] is one of the most widely adopted APR benchmarks, which contains 395 real bug-fix patches from 6 open source GitHub projects in version 1.2, and 835 real bug-fix patches from 17 open source GitHub projects in version 2.0. Each bug-fix example is accompanied with test cases to validate the fix.

5.3.4 VulRD. VulRD [16] merges the datasets Big-Vul [14] and CVEfixes [4]. The Big-Vul dataset contains 3,754 C/C++ code vulnerabilities extracted from GitHub projects (2002-2019), while CVEfixes curates 5,495 vulnerabilities from the National Vulnerability Database (2002-2021). Following the deduplication process, VulRD comprises 6,008 unique vulnerabilities, distributed in a 70-10-20 ratio for training, validation, and testing respectively.

6 Study Result

6.1 Compare with state-of-the-art approaches

6.1.1 Experimental Design. In this section, we aim to evaluate the performance of `Mulpor` on Defects4J [29]. Due to the previous study [12] demonstrates that there exists a common benchmark overfitting in APR evaluation, especially in Defects4J dataset some tools perform better than other benchmarks. Therefore, we adopt Defects4J-v1.2 and Defects4J-v2.0 as evaluation benchmark to mitigate the benchmark overfitting. We consider two settings with the perfect fault localization (FL) and with the spectrum-based FL. Following the prior works [18, 27, 37, 40, 58, 63], we use patch correctness results gathered from previous papers [51, 58, 63] for Defects4J-v1.2 and Defects4J-v2.0 evaluation. We only obtain repair results under specific settings (either perfect or spectrum-based fault localization) for a number of tools (i.e., PraPR, TBar, CURE, Recoder, Repilot, AlphaRepair and GAMMA), and for such tools, we obtained the replication packages or source codes from their respective papers and then rerun them with the absent fault localization strategy. During the reproduction, we maintained the

Table 3: The performance of different approaches on the Defects4J v1.2 and v2.0 dataset (in number of correct repairs).

Tool	Perfect FL		Spectrum-based FL	
	v1.2	v2.0	v1.2	v2.0
PraPR†	-	-	41	7
TBar†	68	8	43	6
CoCoNuT	43	-	-	-
RewardRepair	44	43	27	24
CURE	55	19	38	11
SelfAPR	63	45	39	28
Recoder	65	19	49	9
Repilot	66	50	47	30
RAP-Gen	72	53	48	26
AlphaRepair	74	36	50	20
GAMMA	82	45	47	20
Mulpor	**92**	**59**	**56**	**31**

† PraPR and TBar are template-based tools.
"-" indicates data unavailability.

Table 4: Comparison with state-of-the-art APR techniques with Perfect FL (in number of correct repairs).

Tool	Chart	Closure	Lang	Math	Mockito	Time	Total
TBar	11	16	13	22	3	3	68
CoCoNuT	7	9	7	16	4	1	44
RewardRepair	5	12	7	17	3	1	45
CURE	10	14	9	19	4	1	57
SelfAPR	9	19	10	18	5	2	63
Recoder	10	21	11	18	2	3	65
Repilot	6	22	15	21	0	2	66
RAP-Gen	9	22	12	**26**	2	1	72
AlphaRepair	9	23	13	21	5	3	74
GAMMA	11	24	16	25	3	3	82
Mulpor	**13**	**26**	**19**	23	**7**	**4**	**92**

same settings (e.g., the upper limit of the number of patches) as in the original papers. Note that for CoCoNut, the authors did not provide the trained model, and some training scripts are missing, as reported by some researchers[4]. In the case of PraPR, it currently only supports spectrum-based FL, not perfect FL. To perform perfect FL with PraPR, we need to modify the source code. The author provides the PraPR source code only for Maven projects, while some projects in Defects4J-v1.2 are Gradle projects. Therefore, for PraPR, we only replicate it under the spectrum-based FL setting in Defects4J-v2.0. Unavailability values are indicated by "-" in Table 3.

6.1.2 Results. Table 3 represents the number of bugs that different APR techniques successfully fix on the Defects4J-v1.2 and Defects4J-v2.0 with two FL settings. Overall, we find that Mulpor outperforms compared APR techniques including both traditional and learning-based APR techniques in two FL settings. Specifically, it repairs 10 and 6 more bugs than best baselines in v1.2 and 2.0, respectively. For the results with spectrum-based FL, Mulpor achieves the state-of-the-art performance also, which repairs 6 and 3 more bugs compared with AlphaRepair (50 v.s. 56) and SelfAPR (28 v.s. 31). Besides, Mulpor fixes 13, 26, 19, 23, 7, 4 for Chart, Closure, Lang, Math, Mockito, and Time projects, respectively. Five of these projects exhibit the best performing, as shown in Table 4. Overall, both results with or without perfect FL validate the superiority of our Mulpor over both traditional and learning-based baselines.

[4]https://github.com/lin-tan/CoCoNut-Artifact/issues/2

Table 5: The performance of different approaches on the VulRD dataset (in %).

Approach	PP	PP@1	PP@5	PP@10
VRepair	12.3	7.6	11.2	12.3
SeqTran	20.9	15.4	19.4	20.0
VulRepair	18.1	14.1	17.0	17.8
VulRepair+	22.5	18.2	21.7	22.1
Mulpor	**34.2**	**25.6**	**31.7**	**33.2**

6.1.3 Overlap Analysis. To investigate to what extent Mulpor complements existing APR techniques, we further calculate the number of overlapping bugs fixed by different techniques in Defects4J-v1.2. We select one best-performing traditional technique (i.e., TBar) and three best-performing learning-based techniques (i.e., GAMMA, AlphaRepair, and RAP-Gen) including two cloze-style APR techniques(i.e., GAMMA and AlphaRepair). As shown in Fig. 4, we found that Mulpor is able to fix the most number of unique bugs of 13 that other APR approaches fail to fix, which is 9, 10, 6, and 4 more than TBar, GAMMA, AlphaRepair, and RAP-Gen, respectively. We analyzed the 13 unique bugs fixed by Mulpor and found that the repair granularity of 7 bugs is at the statement level, 4 bugs are at the expression level, and 2 bugs are at the token level. This indicates that Mulpor's design can fix more bugs through multi-granularity patch generation compared to the tools selected for overlap analysis. Specifically, the selected tools, GAMMA, AlphaRepair, and RapGen, are based on UniXcoder, CodeBERT, and CodeT5. These models are pre-trained on a token-level mask language modeling task, which may limit their ability to fix complex bugs. More importantly, compare with other two cloze-style APR techniques, i.e., GAMMA and AlphaRepair, Mulpor can repair 25 and 40 unique bugs, respectively, highlighting the benefits of pre-training model at different granularities. Overall, the result means that Mulpor can be integrated with other techniques to further increase the number of correct patches in Defects4J-v1.2 benchmark.

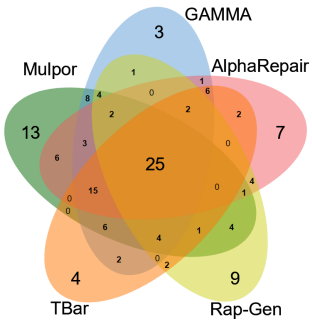

Figure 4: The overlaps of the bugs fixed by different approaches.

6.2 Generalizability of Mulpor

6.2.1 Experimental Design. We have demonstrated that Mulpor achieves impressive performance to repair real-world bugs from the widely-adopted Defects4J benchmark. In order to investigate the Generalizability of Mulpor, we investigate the effectiveness of Mulpor in a different scenario, i.e., vulnerability repair. We report the comparison results with baselines on vulnerability repair in Table 5. Due to there are duplicates in the original dataset and for fair comparison, we rerun the baselines in the deduplicated VulRD dataset. During the inference phase, we use beam search with a beam size of 50 to maintain consistency with baselines.

6.2.2 Results. As shown in Table 5, Mulpor achieves state-of-the-art performance under all metrics by repairing the largest set of vulnerabilities. Specifically, compared with existing state-of-the-art vulnerability repair approach (i.e., VulRepair+), Mulpor can repair

more 52.0% vulnerabilities base on the generated repairs. We also note that at the PP@1 metric, Mulpor achieves a score of 25.6%, while VulRepair+ only achieve 18.2%. This indicates that Mulpor can predict over 40.7% of correct patches at Top-1 prediction, making it a more reliable tool for suggesting repairs to developers.

Table 6: Mulpor's effectiveness on the Top-10 most frequent CWEs.

CWE Type	Name	PP	Proportion
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	29.9	92/308
CWE-125	Out-of-bounds Read	25.0	28/112
CWE-20	Improper Input Validation	26.7	28/105
CWE-264	Permissions, Privileges, and Access Controls	46.0	23/50
CWE-476	NULL Pointer Dereference	40.9	18/44
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	37.2	16/43
CWE-416	Use After Free	35.7	15/42
CWE-190	Integer Overflow or Wraparound	34.2	15/38
CWE-787	Out-of-bounds Write	34.4	11/32
CWE-399	Resource Management Errors	41.9	11/31
Total		31.9	257/805

6.2.3 Vulnerability Type Analysis. CWE (Common Weakness Enumeration) serves as a comprehensive catalog enumerating vulnerability weaknesses in software, highlighting potential security issues with varying degrees of severity. This enumeration offers valuable guidance to organizations and security analysts, aiding them in fortifying their software systems against potential threats. To evaluate the practical implications of Mulpor in real-world scenarios, we conducted a thorough investigation into the Top-10 most frequent CWEs, assessing its efficacy in repairing these commonly occurring weaknesses. These vulnerabilities, representing the most frequent occurrences in our dataset, pose significant risks due to their inherent ease of discovery and exploitation. Exploitation of these weaknesses could potentially empower adversaries to take full control of a system, pilfer data, or disrupt an application's functionality. Table 6 presents an overview of the effectiveness of Mulpor in addressing the Top-10 most frequent vulnerabilities. Notably, Mulpor achieves Perfect Predictions at rates as high as 46.0% for CWE-264, 41.9% for CWE-399, and 40.9% for CWE-476. However, its performance varies for other CWEs, such as CWE-125 and CWE-20, where Mulpor can only repair 25.0% and 26.7% of vulnerabilities, respectively. Further analysis reveals that the median vulnerability lengths for CWE-125 and CWE-20 are 485 and 335, respectively. While Mulpor achieves a 46.0% repair rate for CWE-264, it is important to note that the median length of vulnerabilities in this category is 211. This indicates a decrease in Mulpor's effectiveness as the length of the vulnerable program increases, as discussed in Section 6.2.4, resulting in poorer performance for CWE-20. It is noteworthy that Mulpor shows the ability to fix vulnerabilities perfectly at a rate of 31.9% for the Top-10 most frequent CWEs, compared to 34.2% across the entire dataset. This highlights the strong generalization capabilities of Mulpor, showcasing its proficiency in addressing infrequent CWEs.

6.2.4 Vulnerability Length Analysis. While Mulpor demonstrates proficiency in generating vulnerability repairs for a substantial number of CWEs, it encounters challenges in accurately generating repairs for a considerable number of vulnerable functions. To delve deeper into this issue, we conducted an investigation to analyze Mulpor's performance concerning vulnerability length, including the number of buggy lines. Table 7 presents the performance of Mulpor across different vulnerability lengths. Our findings reveal that the effectiveness of Mulpor is contingent on the size and complexity of vulnerable functions. Specifically, for vulnerabilities with fewer than 100 tokens, Mulpor achieves a PP of 41.3%. However, this rate significantly decreases to 20.2% for vulnerabilities exceeding 500 tokens. Two primary reasons contribute to this decline: (1) As vulnerability lengths increase, the average number of buggy lines also rises, indicating an escalating difficulty in repair. (2) The window size of T5 architecture is limited to 512 tokens that constrains the performance of Mulpor. For vulnerable functions surpassing 512 tokens, any extra tokens are truncated and left unprocessed, resulting in a detrimental impact on the performance of Mulpor.

Table 7: The performance of Mulpor with different lengths of vulnerability (in PP).

Length	Percentage	Avg.†	VulRepair+	Mulpor
1-100	19.3	1.67	41.3	47.2
101-200	19.0	2.32	25.3	38.2
201-300	14.1	2.31	25.8	40.8
301-400	11.6	2.85	20.3	31.7
401-500	7.2	3.78	23.9	21.8
>501	28.9	3.77	11.7	20.2

† Average number of buggy lines

Table 8: Results of the ablation study (in %).

Model	−SG	−EG	−TG	−RR	Mulpor
PP@1	23.5	22.7	20.4	23.4	**25.6**
PP	28.7	29.4	27.6	**34.2**	**34.2**

Additionally, we explore the performance of existing state-of-the-art technique (i.e., VulRepair+) across varying vulnerability lengths. Remarkably, vulnerability length exerted a more pronounced negative impact on VulRepair+ compared to Mulpor. Specifically, VulRepair+ can only repair 11.7% of vulnerabilities exceeding 500 tokens, whereas Mulpor achieves a PP of 20.2%. This discrepancy highlights the relative resilience of Mulpor in handling longer vulnerabilities compared to existing techniques.

6.3 Ablation study

We have demonstrated that well-designed multi-granularity generator play a crucial role in enabling Mulpor to achieve state-of-the-art performance across various APR scenarios. To further understand the contribution of each component, we conducted an investigation by creating four variants of Mulpor, each with one critical component removed, and subsequently evaluated their performance on the similar downstream tasks. Note that we opted to conduct the ablation study on vulnerability repair, as its dataset is larger than that used for bug repair. This can reduce randomness and more accurately reflect the contribution of each component.

Overall, the contributions of the statement-level generator (SG), expression-level generator (EG), token-level generator (TG), and patch re-ranking (RR) to the performance of Mulpor are evident, as depicted in Table 8. Notably, the exclusion of the TG leads to the most significant decrease in PP@1 and PP for Mulpor, by 20.3% and 19.3%, respectively. This aligns with our expectations, as the patch generation process involves sequential token generation. The TMP task aids the token-level generator in comprehending token distribution in code snippets, facilitating the generation of accurate

patches. The PP also experiences a noticeable decrease of 16.1% when SG is excluded, highlighting that statement-level masking enhances the model's understanding of code syntactics and semantics. Furthermore, without the EG, the PP of Mulpor decreases by 14.0%. This aligns with our expectations, as the granularity of expressions lies between statements and tokens, allowing Mulpor to compensate for the absence of EG with contributions from other generators. It is noteworthy that the decrease in PP@1 caused by SG is more pronounced than that caused by EG. We attribute this to the larger search space of SG compared to EG, making it more challenging for SG to generate the correct patch at the top-1 compared to EG. Additionally, the exclusion of the patch re-ranking process results in an 8.6% decrease in PP@1, emphasizing the effectiveness of patch re-ranking in enhancing the overall performance of Mulpor.

7 Discussion

7.1 Efficiency of Mulpor

Running on a single NVIDIA GeForce RTX 4090 GPU, Mulpor can generate 15.5 patches per second. This means that Mulpor can generate patches for a given snippet of buggy code in the Defects4J-v1.2 dataset in an average of 6 seconds. By comparison, the state-of-the-art technique, GAMMA, can generate 20.7 patches per second, but it produces more bugs than Mulpor and takes about 23 seconds for each bug. Mulpor is faster since it introduces fewer patches due to its more precise search space. As a result, the beam size of Mulpor can be set to a value which is much smaller than that of GAMMA (15 vs. 250).

7.2 Threats to Validity

Internal Threats. Our first internal threat comes from the manual validation of patch correctness. To mitigate the influence of potential bias, we follow the previous studies [27, 58, 62, 63], first three authors manually inspect all patches that pass all test cases. A plausible patch is identified as a correct patch if first three authors identify it as semantically equivalent to a ground truth patch. When evaluating the effectiveness of Mulpor for general repairs, we only consider the first plausible patch after ranking, as done in previous studies [37, 64]. To facilitate replication and verification of our experiments, we have made the relevant materials publicly available in our online repository.

The second internal threat involves potential data leakage from pre-trained models. In our experiment, we initialize the parameters of Mulpor with weights from CodeT5, and then further train it on three different pre-training tasks before evaluating it on a downstream tasks. Considering the pre-training dataset includes 2.3M functions from CodeSearchNet, we first check for functions in the pre-training dataset and then remove all duplicates before the pre-training phase of Mulpor. After the pre-training phase of Mulpor, we expect that the leaked knowledge from CodeT5 will be somewhat forgotten. Besides, we manually perturb all 10 leaked buggy codes (by changing variable names, adding dead code following the previous studies [58, 63]) in Defects4J-v1.2. We find that Mulpor can still generate correct patches for all bugs. This suggests that Mulpor is not merely overfitting to patches present in the original CodeT5 training dataset. Thus, we are confident that the data leakage is not a key point to our conclusion.

The third internal threat stems from the reuse of baseline results from previous studies, specifically within the Defects4J benchmark. This benchmark is a widely-used benchmark in the APR community, and many studies use it to assess the effectiveness of their proposed approaches. Therefore, it is a common practice in the APR community [27, 40, 51, 62, 63] by reusing the results of baselines on Defects4J from previous studies. To ensure a fair comparison, we thoroughly review the settings in the baselines and strive to maintain consistency. However, some settings do differ. For instance, while most learning-based techniques [34, 58, 63, 64] have a 5-hour time limit, TBar has a 3-hour limit. In such cases, we set the running-time limit to the lowest value used in the baselines. Additionally, the machines used in baselines also vary. In learning-based techniques, the performance of the machines only has limited impact on the techniques' performance. This is because the model weights in these techniques are fixed, and only time efficiency may be affected. For instance, when running on a single NVIDIA GeForce RTX 4090 GPU, the GAMMA takes about 23 seconds to generate patches for each bug, while the running-time limit for each bug is five hours, which means the inference procedure can be completely finished within the time limit. On the other hand, different machines could yield different results in template-based techniques due to the supported search space and time constraints. To mitigate potential bias, we reran TBar in Defects4J-v2.0 and achieved the same results as reported in previous studies [58, 60, 64]. Therefore, the impact of different machines used in baselines is minimal.

External Threats. The primary external threats to validity arise from the choice of evaluation benchmarks. The performance claims made for Mulpor may not seamlessly extend to other datasets. To mitigate this threat, we assess the generalizability of Mulpor by conducting evaluations on Defects4J-v1.2 and Defects4J-v2.0. Furthermore, we evaluate our claims about the generalization to other code repair scenarios by investigating the performance of Mulpor in vulnerability repair. The results affirm that Mulpor exhibits generalizability across distinct datasets and scenarios.

8 CONCLUSION

This paper introduces Mulpor, which is a simple and effective multi-granularity patch generation approach designed to address the diverse nature of real-world software bugs. Mulpor initially acquires semantic and syntactic knowledge through unsupervised pre-training tasks at different granularities. Subsequently, it undergoes fine-tuning on real-world bug-fixing datasets. This research emphasizes the critical importance of generating patches from multiple granularities for program repair techniques. By doing so, Mulpor demonstrates promising results in achieving effective bug-fixing performances across various program repair scenarios, including bug repair and vulnerability repair. All data in this study are publicly available at: **https://zenodo.org/records/12660892**.

Acknowledgments

This work is supported by the National Key R&D Program of China (No. 2022YFA1005101), the National Natural Science Foundation of China (No.62372193), and the Research Foundation from NUDT (Grant No. ZK24-05).

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*. IEEE, 89–98. https://doi.org/10.1109/TAIC.PART.2007.13

[2] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 907–918.

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

[4] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.

[5] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. GZoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 378–381. https://doi.org/10.1145/2351676.2351752

[6] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: Generative Pre-Training by "naturalizing" Source Code *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 18–30. https://doi.org/10.1145/3540250.3549162

[7] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.

[8] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. on Software Engineering* (2019).

[9] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2022. Seqtrans: automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering* 49, 2 (2022), 564–585.

[10] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259* (2014).

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

[12] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 302–313. https://doi.org/10.1145/3338906.3338911

[13] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, 313–324. https://doi.org/10.1145/2642937.2642982

[14] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[16] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947. https://doi.org/10.1145/3540250.3549098

[17] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[18] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 19–30. https://doi.org/10.1145/3293882.3330559

[19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[20] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*.

[22] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. AAAI, 1345–1351.

[23] Dan Hao, Lu Zhang, Lei Zang, Yanbo Wang, Xingxia Wu, and Tao Xie. 2015. To be optimal or not in test-case prioritization. *IEEE Transactions on Software Engineering* 42, 5 (2015), 490–505.

[24] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-tuning Large Language Models of Code for Automated Program Repair. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

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

[26] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 298–309. https://doi.org/10.1145/3213846.3213871

[27] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[28] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. 2020. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the association for computational linguistics* 8 (2020), 64–77.

[29] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis*. ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[30] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25, 3 (2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[31] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 593–604. https://doi.org/10.1145/3106237.3106309

[32] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. 213–224. https://doi.org/10.1109/SANER.2016.76

[33] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[34] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

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

[36] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. 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 (ESEC/FSE 2022)*. 1035–1047. https://doi.org/10.1145/3540250.3549081

[37] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 31–42. https://doi.org/10.1145/3293882.3330577

[38] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 118–129. https://doi.org/10.1109/SANER.2018.8330202

[39] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 166–178. https://doi.org/10.1145/2786805.2786811

[40] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. 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*. 101–114. https://doi.org/10.1145/3395363.3397369

[41] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 691–701. https://doi.org/10.1145/2884781.2884807

[42] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*. 1169–1180.

[43] Martin Monperrus. 2018. Automatic software repair: A bibliography. *Comput. Surveys* 51, 1 (2018), 17:1–17:24. https://doi.org/10.1145/3105906

[44] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[45] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 254–265. https://doi.org/10.1145/2568225.2568254

[46] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[47] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21 (2020), 1–67.

[48] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 130–140. https://doi.org/10.1109/SANER.2018.8330203

[49] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology* 28, 4 (2019), 19:1–19:29. https://doi.org/10.1145/3340544

[50] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far are We?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM.

[51] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair. *arXiv preprint arXiv:2309.06057* (2023).

[52] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. 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*. 8696–8708.

[53] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184.

[54] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[55] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1–11. https://doi.org/10.1145/3180155.3180233

[56] Alexander Wettig, Tianyu Gao, Zexuan Zhong, and Danqi Chen. 2022. Should you mask 15% in masked language modeling? *arXiv preprint arXiv:2202.08005* (2022).

[57] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 479–490. https://doi.org/10.1109/SANER.2019.8668043

[58] Chunqiu Steven Xia and Lingming Zhang. 2022. 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*. 959–971.

[59] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[60] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[61] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*. 1506–1518.

[62] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: Continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 678–690.

[63] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting template-based automated program repair via mask prediction. *arXiv preprint arXiv:2309.09308* (2023).

[64] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. 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*. 341–353.

Received 2024-04-12; accepted 2024-07-03