Learning to Construct Better Mutation Faults

Zhao Tian
College of Intelligence and Computing, Tianjin University
Tianjin, China
tianzhao@tju.edu.cn

Junjie Chen*
College of Intelligence and Computing, Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Qihao Zhu
Key Laboratory of HCST, MoE
DCST, Peking University
Beijing, China
Zhuqh@pku.edu.cn

Junjie Yang
College of Intelligence and Computing, Tianjin University
Tianjin, China
jjyang@tju.edu.cn

Lingming Zhang
University of Illinois at Urbana-Champaign
Urbana, IL, USA
lingming@illinois.edu

ABSTRACT

Mutation faults are the core of mutation testing and have been widely used in many other software testing and debugging tasks. Hence, constructing high-quality mutation faults is critical. There are many traditional mutation techniques that construct syntactic mutation faults based on a limited set of manually-defined mutation operators. To improve them, the state-of-the-art deep-learning (DL) based technique (i.e., DeepMutation) has been proposed to construct mutation faults by learning from real faults via classic sequence-to-sequence neural machine translation (NMT). However, its performance is not satisfactory since it cannot ensure syntactic correctness of constructed mutation faults and suffers from the effectiveness issue due to the huge search space and limited features by simply treating each targeted method as a token stream.

In this work, we propose a novel DL-based mutation technique (i.e., LEAM) to overcome the limitations of both traditional techniques and DeepMutation. LEAM adapts the syntax-guided encoder-decoder architecture by extending a set of grammar rules specific to our mutation task, to guarantee syntactic correctness of constructed mutation faults. Instead of predicting a sequence of tokens one by one to form a whole mutated method, it predicts the statements to be mutated under the context of the targeted method to reduce search space, and then predicts grammar rules for mutation fault construction based on both semantic and structural features in AST. We conducted an extensive study to evaluate LEAM based on the widely-used Defects4J benchmark. The results demonstrate that the mutation faults constructed by LEAM can not only better represent real faults than two state-of-the-art traditional techniques (i.e., Major and PIT) and DeepMutation, but also substantially boost two important downstream applications of mutation faults, i.e., test case prioritization and fault localization.

*Junjie Chen is the corresponding author.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; **• Computing methodologies → Artificial intelligence**.

KEYWORDS

Mutation Testing, Deep Learning, Fault Injection

ACM Reference Format:
Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2022. Learning to Construct Better Mutation Faults. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556949

1 INTRODUCTION

Mutation faults are originally proposed in mutation testing, which aim to mimic real faults and then measure the effectiveness of a test suite [35, 36, 62]. Specifically, by applying a series of mutation operators (e.g., relational operator replacement) to the program under test, a set of mutation faults can be constructed. Then, the effectiveness of a test suite can be measured by executing the test suite to detect the set of mutation faults. Undoubtedly, mutation faults are the core of mutation testing. Indeed, as demonstrated by the existing studies [48, 61], the quality of mutation faults can significantly affect the effectiveness of mutation testing. Besides measuring test effectiveness, mutation faults have been extensively extended to facilitate many other software testing and debugging tasks (also called downstream applications of mutation faults), e.g., test case prioritization (TCP) [55, 70] and fault localization (FL) [57, 63]. In fact, mutation-based techniques have become the state of the art for various such downstream applications. Hence, due to the important role and wide range of usage scenarios, constructing high-quality mutation faults has become more and more critical.

Over the years, many techniques have been proposed to construct mutation faults in order to represent real faults as much as possible [21, 34, 40, 56, 58, 69, 76]. For ease of presentation, we call a technique constructing mutation faults *a mutation technique* in our paper. The traditional mutation techniques construct mutation faults by manually designing a series of mutation operators, each of which can conduct a simple syntactic change to the program under mutation for creating a mutation fault [21, 34, 40, 56, 58, 69]. Despite simple, some of them have achieved good effectiveness and have been used in both practice and academia, e.g., Major [40] and PIT [21]. However, as revealed by the existing studies [12, 28, 65, 76, 77], these mutation faults constructed by the current set of mutation operators cannot sufficiently represent real faults. There are two major reasons: (1) The current set of mutation operators is limited. Designing more mutation operators may be helpful to relieve this problem, but it requires substantial manual efforts. (2) The current set of mutation operators focuses on simple syntactic changes, indicating that it is hard to go deep into fault semantics. Hence, they cannot construct the mutation faults specific to the semantics of the program under mutation.

To relieve the limitations of traditional mutation techniques, some existing work suggested to extract mutation operators from historical bugs/fixes [12, 65]. Same as this idea, Tufano et al. [77] proposed the state-of-the-art *deep learning* (DL) based mutation technique (i.e., DeepMutation), which constructs mutation faults by learning from a number of real faults via classic sequence-to-sequence neural machine translation (NMT). Indeed, DeepMutation opens a direction to avoid the efforts of manually designing mutation operators and incorporate program semantics via deep learning, but it was just evaluated in terms of DL metrics and thus it is unclear whether its constructed mutation faults can really improve mutation testing and its downstream applications (e.g., mutation-based TCP). From our study (to be presented in Section 5), we demonstrate for the first time that DeepMutation actually underperforms the traditional techniques (i.e., Major and PIT) in the usage scenarios of mutation faults. That is, despite novel, DeepMutation does not reach the requirement of practicality like Major and PIT. The main reasons are threefold: (1) It constructs a mutation fault (i.e., a mutated method) by predicting a sequence of tokens one by one in the method, which constitutes huge search space, and thus it is hard to achieve accurate prediction to form an expected mutated method. (2) It treats a method to be mutated as a token stream, which actually loses much program information, and thus the prediction performance can be negatively affected. (3) It cannot ensure to produce syntactically correct programs after mutation.

To overcome the limitations of the state-of-the-art DL-based technique, in this work, we propose a novel DL-based mutation technique, called **LEAM** (**LEA**rning to **M**utate), by designing a syntax-guided mutation process inspired by the existing neural program generation techniques [73, 74, 89]. It aims to construct better mutation faults for facilitating both mutation testing and the downstream applications of mutation faults. To reduce the search space when constructing a mutation fault (overcoming the first limitation), LEAM builds a sub-model for predicting the statements to be mutated under the context of the targeted method, rather than directly predicting a sequence of tokens one by one to form a whole mutated method. To improve the prediction performance (overcoming the second limitation), LEAM transforms a targeted method as an AST (rather than a token stream) for DL model building in order to incorporate both structural and semantic information. To guarantee the syntactic correctness of constructed mutation faults (overcoming the third limitation), LEAM adapts the syntax-guided encoder-decoder architecture and builds another sub-model for predicting a grammar rule for each unexpanded non-terminal node in the partial AST corresponding to the identified statements to be mutated. By integrating these sub-models built based on AST information, LEAM constructs mutation faults for a targeted method. In particular, since a method may introduce different faults, LEAM incorporates the beam search algorithm [26, 43] for constructing *a set of* mutation faults that are highly possible to occur in practice, for a targeted method.

To evaluate the effectiveness of LEAM, we conducted an extensive study based on the widely-used Defects4J benchmark [41] by comparing with two typical traditional mutation techniques (i.e., Major [40] and PIT [21]) and the state-of-the-art DL-based mutation technique (i.e., DeepMutation [77]). Specifically, we compared the quality of mutation faults constructed by the four techniques in three popular scenarios (i.e., mutation testing, and its two important downstream applications – mutation-based TCP and FL). Our experimental results show that LEAM can construct more representative mutation faults than the three compared techniques in all the three scenarios. For example, in mutation testing, the mutation faults constructed by LEAM can better represent real faults and the mutation faults constructed by other mutation techniques. In mutation-based TCP, feeding the mutation faults constructed by LEAM to the state-of-the-art mutation-based TCP techniques (i.e., GRK, GRD, and HYB-ω [70]) achieves 15.38%~280.00% improvements in terms of average TCP effectiveness than the three compared techniques. In mutation-based FL, feeding the mutation faults constructed by LEAM to the state-of-the-art mutation-based FL techniques (i.e., MUSE [57] and Metallaxis [63]) achieves 110.71%~600.00% improvements in terms of average Top-1 FL effectiveness than the three compared techniques.

To sum up, our work makes the following main contributions:

- We propose a novel DL-based mutation technique (LEAM), which adapts the syntax-guided encoder-decoder architecture to build two sub-models based on AST information, for better learning to represent real faults and ensuring syntactic correctness of mutation faults.
- We conduct an extensive study to evaluate LEAM in three popular scenarios, including mutation testing and its two downstream applications (mutation-based TCP and FL). The results demonstrate the significant superiority of LEAM over two traditional techniques and the state-of-the-art DL-based technique in all the three scenarios.
- We develop and release our tool and the built model for promoting future research and practical use. Please find them at: **https://github.com/tianzhaotju/LEAM**.

2 BACKGROUND

For ease of understanding, in this section we first introduce some basic concepts on mutation testing and mutation faults in Section 2.1. Then, we introduce two important downstream applications of mutation faults, i.e., mutation-based TCP (Section 2.2) and mutation-based FL (Section 2.3). This is because besides the original usage scenario (i.e., mutation testing), mutation faults have been widely used in many other testing and debugging tasks. To more sufficiently evaluate the effectiveness of LEAM, we also investigate the quality of constructed mutation faults in these downstream applications by taking the two as the representative (Section 4).

2.1 Mutation Testing and Mutation Faults

Mutation testing aims to reveal a set of deliberately introduced faults with regard to a program under test, in order to measure the effectiveness of a test suite (and further augment the test suite) [36]. Its basic assumption is that the introduced faults can effectively represent real faults [5, 6, 22]. Here, a program with an introduced fault is called a **mutation fault** (or *mutant*), which is constructed by deliberately changing a small portion of code in the program under test. The changing rules are called **mutation operators**. For example, given that "`if(x>y)`" is a code fragment in the original program under test, a mutation fault can be constructed by changing it into "`if(x<y)`" through the mutation operator of relational operator replacement. Please note that different mutation techniques may include different sets of mutation operators.

Based on a set of constructed mutation faults, the effectiveness of a test suite can be measured by running each test case on each mutation fault. If the original program and a mutation fault produce different outputs after the execution of a test case, we say that the mutation fault is **killed** by the test case. If a mutation fault cannot be killed by the whole test suite, we say that the mutation fault is **live** with regard to the test suite. In particular, there are mutation faults equivalent to the original program, and thus they cannot be killed by any test cases (not only the test cases in the test suite). These mutation faults are called **equivalent mutation faults**. By computing the **mutation score**, which is the ratio of the number of killed mutation faults by the test suite to the total number of constructed mutation faults (except equivalent mutation faults), the effectiveness of the test suite can be measured.

Mutation testing is one of the most effective ways of measuring test effectiveness [9, 20, 86], and its core lies in mutation faults [36, 65, 77]. Therefore, constructing high-quality mutation faults for better representing real faults is a critical task.

2.2 Mutation-based TCP

Test case prioritization (TCP) aims to schedule the execution order of test cases for detecting faults earlier, which has been widely studied in the literature [13, 14, 18, 31, 68, 80]. Over the years, a large number of TCP techniques have been proposed, such as coverage-based TCP [16, 29, 31, 64, 88] and mutation-based TCP [25, 55, 70]. As demonstrated by the existing studies [55, 70], mutation-based TCP techniques achieve better prioritization effectiveness than the most widely-studied coverage-based TCP techniques, and thus have become one of the mainstream TCP techniques. In our study, we will compare different mutation techniques by feeding their constructed mutation faults to a mutation-based TCP technique respectively, and then analyze the corresponding achieved TCP effectiveness. Next, we briefly introduce three state-of-the-art mutation-based TCP techniques, which are also the ones used in our study.

GRK iteratively selects a test case that maximizes the number of additionally *killed* mutation faults [70]. It aims to distinguish the mutation faults from the original program as early as possible. **GRD** iteratively selects a test case that maximizes the number of additionally *distinguished* mutation faults [70]. Here, mutation faults are distinguished by a test case when their outputs are different after the execution of the test case. That is, GRD aims to distinguish all the mutation faults from each other as early as possible. **HYB-ω** combines both GRK and GRD, which iteratively selects a test case that maximizes the weighted sum of the number of additionally killed mutation faults and the number of additionally distinguished mutation faults. When the weight of the former (denoted as $\omega \in [0, 1]$) is 1, HYB-ω is equivalent to GRK; when $\omega = 0$, HYB-ω is equivalent to GRD. In our study, we set $\omega = 0.5$ in HYB-ω following the existing work [70].

2.3 Mutation-based FL

Fault localization (FL) aims to automatically localize faulty program elements (e.g., statements or methods) by ranking all the program elements based on their suspicious scores, which tend to be computed based on various dynamic execution information. FL have received extensive attention over the years [3, 15, 17, 39, 45, 57, 67, 81], and largely promoted the development of its follow-up task (i.e., automated program repair) [7, 27, 59, 83, 89]. In the literature, a large number of FL techniques have been proposed, such as spectrum-based FL [3, 39, 45] and mutation-based FL [57, 63]. Indeed, mutation-based FL techniques are one kind of the most widely-studied FL techniques, and their effectiveness has been demonstrated by the existing studies [32, 57, 63, 66]. In our study, we will also compare different mutation techniques by investigating the effectiveness of a mutation-based FL technique by feeding the constructed mutation faults by these mutation techniques to it respectively. Here, we brief introduce two state-of-the-art mutation-based FL techniques, which are also the ones used in our study.

MUSE [57] and **Metallaxis** [63] are two state-of-the-art mutation-based FL techniques. In general, mutation-based FL techniques consider whether the execution of a statement affects the result of a test case by injecting mutation faults. If a statement affects failing test cases more frequently but affects passing test cases more rarely, it is more suspicious. The main difference between MUSE and Metallaxis lies in how to utilize mutation faults to compute the suspicious score of each statement. MUSE first computes the suspicious score of each mutation fault as shown in Formula 1

$$S(m) = failed(m) - \frac{f2p}{p2f} \cdot passed(m) \tag{1}$$

where $failed(m)/passed(m)$ is the number of test cases that fail/pass on the original program but pass/fail on the mutation fault m, $f2p/p2f$ is the number of test cases that change from "fail"/"pass" to "pass"/"fail" on any mutation fault. Then, the suspicious score of a statement (denoted as s) is the average of the suspicious scores of all the mutation faults occurring at s.

Metallaxis computes the suspicious score of each mutation fault as shown in Formula 2.

$$S(m) = \frac{failed(m)}{\sqrt{totalfailed \cdot (failed(m) + passed(m))}} \tag{2}$$

where *totalfailed* is total number of test cases that fail on the original program, $failed(m)$ is the number of test cases that fail on the original program but the output changes on the mutation fault m, and similarly for $passed(m)$. Then, the suspicious score of a statement (denoted as s) is the maximum of the suspicious scores of all the mutation faults occurring at s.

Following the existing studies [8, 50–52], we used the two techniques for localizing potential faulty *methods* (i.e., method-level

FL) in our study. Therefore, for MUSE and Metallaxis, we further computed the suspicious score of each method by using the maximum of the suspicious scores of all the statements in the method following the existing work [51, 52, 72].

3 APPROACH

We propose a novel DL-based mutation technique by learning from a large number of real faults, called **LEAM**, which can effectively overcome the limitations of traditional mutation techniques (i.e., requiring substantial manual efforts to design mutation operators for representing real faults but still failing to construct the faults specific to the semantics of the program under mutation). Similar to the state-of-the-art DL-based mutation technique (i.e., DeepMutation) [77], LEAM conducts mutation at the method granularity, i.e., constructing mutated methods for a targeted method in the program. However, different from it, LEAM has the following significant advantages to construct better mutation faults:

- *Guaranteed syntactic correctness.* LEAM adapts the syntax-guided encoder-decoder architecture for mutation fault construction by extending a set of grammar rules specific to our task. In this way, a mutation fault can be constructed by applying a sequence of predicted grammar rules to change the targeted method, and thus it is syntactically correct.
- *More comprehensive program features.* Instead of treating a method to be mutated as a token stream, LEAM extracts both semantic and structural features from the method to be mutated by representing it as an AST. Such comprehensive features are helpful to build a more accurate model.
- *Reduced search space.* Instead of predicting a sequence of tokens one by one to form a whole mutated method, LEAM first predicts the statements highly possible to be mutated under the context of the targeted method, which is enabled by adding the corresponding grammar rules and can largely reduce mutation space, and then predicts a sequence of grammar rules for changing the statements.

As a method may introduce different faults, LEAM incorporates the beam search algorithm [26, 71] for constructing a set of mutation faults highly possible to occur in practice for the targeted method. Figure 1 shows the overall architecture of LEAM. In the following, we will introduce the set of our extended grammar rules in LEAM in Section 3.1, our extracted features for model building in Section 3.2, our DL model for grammar rule prediction (including statement prediction and change prediction) in Section 3.3, and the beam-search-based mutation fault construction process in Section 3.4.

3.1 Grammar Rule Definition

Inspired by neural program generation [73, 74, 89], LEAM adapts the syntax-guided encoder-decoder architecture for our task of mutation fault construction in order to guarantee syntactic correctness of each constructed mutation fault. Specifically, it aims to predict the probability of each grammar rule for expanding an unexpanded non-terminal node in a partial AST. Please note that the mutation process is conducted at the AST level in LEAM since (1) this level can well support the syntax-guided architecture and (2) an AST can provide more comprehensive information for model building (to be presented in Section 3.2). Through applying a sequence of predicted grammar rules to expand from the start symbol, a syntactically correct mutation fault can be constructed.

Table 1: Definition of extended grammar rules for our task

1. Start	$\longrightarrow$ IdentifiedStmts
2. IdentifiedStmts	$\longrightarrow$ IdentifiedStmt; IdentifiedStmts | end
3. IdentifiedStmt	$\longrightarrow$ Insert | Modify | Delete
4. Insert	$\longrightarrow$ insert($\langle NTSstmt \rangle$)
5. Modify	$\longrightarrow$ modify($\langle ID \rangle$, $\langle NTS \rangle$)
6. Delete	$\longrightarrow$ delete($\langle IDstmt \rangle$)
7. $\langle NTS \rangle$	$\longrightarrow$ $\langle GRS \rangle$
8. $\langle Identifier \rangle$	$\longrightarrow$ identifier | placeholder

* $\langle NTS \rangle$ stands for a non-terminal symbol.
* $\langle NTSstmt \rangle$ belongs to $\langle NTS \rangle$, but refers in particular to the non-terminal in the grammar of the targeted programming language representing a statement.
* $\langle ID \rangle$ refers to the ID of an AST node representing a NTS.
* $\langle IDstmt \rangle$ belongs to $\langle ID \rangle$, but refers in particular to the ID of the root node for the identified statement.
* $\langle GRS \rangle$ refers to the grammar rules defined in the targeted programming language.
* $\langle Identifier \rangle$ is the non-terminal in the grammar of the targeted programming language representing an identifier.

Here, the set of grammar rules to be predicted in LEAM contains two subsets: (1) all the grammar rules defined in the targeted programming language of the program under mutation, which is the key to guarantee syntactic correctness of each constructed mutation fault; (2) our extended grammar rules to enable the architecture to support our task of mutation fault construction. Since the former subset of grammar rules is defined by the targeted programming language, we just present the definition of our extended grammar rules for our task in Table 1. Please note that LEAM is a general technique and can be applied to any programming language with the concept similar to `statement`, and thus we define these grammar rules in a general manner.

Instead of predicting a sequence of tokens one by one to form a whole mutated method, LEAM reduces the search space by first predicting the statements highly possible to be mutated under the context of the targeted method. Then, the mutation process can be just conducted on the identified statements rather than the whole method. Specifically, LEAM defines *Rules 1-2* to support the prediction of the statements to be mutated. From the two rules, LEAM supports both single-statement mutation and multiple-statements mutation, which could represent real faults better than traditional mutation techniques (e.g., Major [40] and PIT [21]) that only conduct one syntactic change on one statement in each mutation.

Rules 3-6 define three operations on each identified statement. The `insert` operation aims to insert a newly generated statement before the identified statement. The parameter `<NTS>` can be expanded to the whole statement to be inserted via a sequence of grammar rules (defined in the targeted programming language as shown in Rule 7). The `modify` operation aims to replace an AST subtree in the identified statement with a new AST subtree. It has two parameters: (1) the ID of the root node for the AST subtree to be replaced, where the ID is defined as the order of a node in the preorder traversal sequence for the AST; (2) the non-terminal to be expanded to the new AST subtree via a sequence of grammar rules,

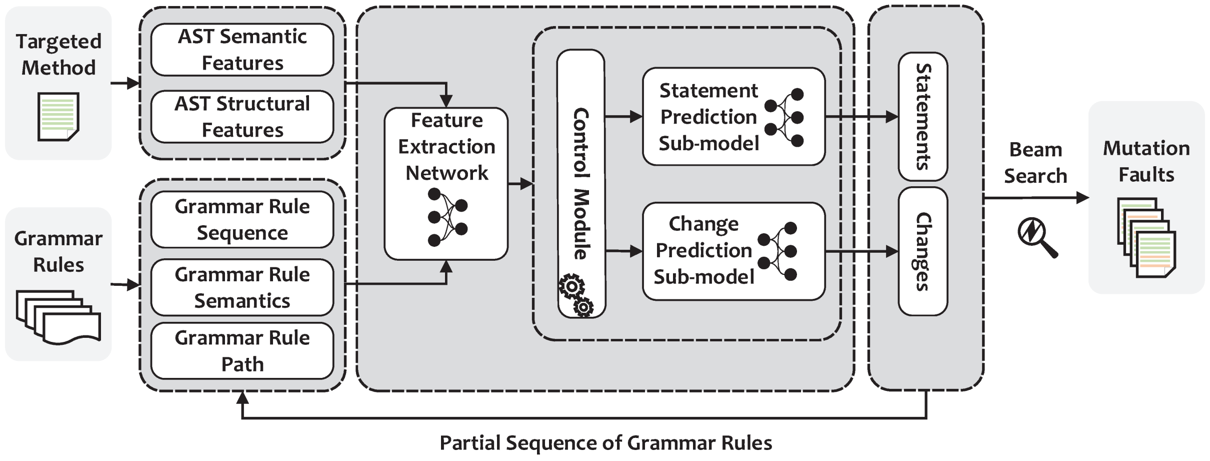

Figure 1: Overview of LEAM

which should be the same as the root node in the first parameter so as to guarantee syntactic correctness after replacement. The `delete` operation aims to delete the identified statement.

To enable the construction of program-specific mutation faults (i.e., generating program-specific identifiers during mutation), LEAM introduces `placeholder` to the grammar of the targeted programming language and changes identifier nodes into non-terminals, as defined in *Rule 8*. Specifically, LEAM replaces infrequent identifiers in the training data as `placeholder` following the existing work [89], and thus enables an identifier node to be expanded to either a `placeholder` or a frequent identifier in training data. Following the existing work [89], when the number of occurring times for an identifier is more than 100 in training data, we regard it as a frequent identifier. When constructing a mutation fault, each `placeholder` can be replaced with any program-specific identifier that can be accessible from the local context and does not incur type errors. Here, LEAM randomly selects one from the program-specific identifiers satisfying the constraints.

3.2 Feature Extraction

LEAM constructs mutation faults by building a DL model for predicting grammar rules. Actually, our DL model contains two sub-models for two-step prediction, i.e., (1) predicting the statements to be mutated in the targeted method based on *Rules 1-2* (called *statement prediction*), and (2) predicting the changes on the identified statements based on *Rules 3-8* and the grammar rules defined in the targeted programming languages (called *change prediction*). Here, we introduce our extracted features for the two sub-models in LEAM. As presented before, LEAM conducts mutation at the method granularity, and indeed each training instance in our training data is a pair of correct method and faulty method (more details about our training data can be found in Section 4.1). Therefore, LEAM extracts features in the scope of the targeted method. Specifically, LEAM extracts both *semantic* and *structural* features by representing the method as an AST rather than a token stream. Since change prediction is based on statement prediction, more features can be extracted for change prediction after statement prediction.

LEAM extracts two kinds of features for statement prediction at the AST level: (1) *AST semantic features*: LEAM first conducts the preorder traversal of the AST to produce a sequence of nodes, and then extracts the semantics of each node as a vector, denoted as $(c_1, c_2, \ldots, c_L)$, where c_i is embedded via word embedding and L is the node sequence length. (2) *AST structural features*: Following the existing work [73, 74, 89], LEAM transforms an AST into a directed graph to capture the structural relations between AST nodes, where the nodes refer to AST nodes and the directed edges are from a node to its children and left sibling. Then, the structural information is represented as an adjacent matrix denoted as $\mathbb{G}^{L\times L}$. These features are further processed by self-attention (ϕ_{self}) and tree convolution (ϕ_{conv}) layers as shown in Formula 3.

$$\begin{aligned} e_1, e_2, \ldots, e_L &= \phi_{self}([c_1, c_2, \ldots, c_L]) \\ a_1, a_2, \ldots, a_L &= \phi_{conv}([e_1, e_2, \ldots, e_L] \times \mathbb{G}^{L\times L}) \end{aligned} \tag{3}$$

Besides the two kinds of features, LEAM additionally extracts three kinds of features for change prediction. Since LEAM is based on the syntax-guided encoder-decoder architecture, the partial sequence of grammar rules that have been predicted also affects the prediction of the next grammar rule. Therefore, LEAM also extracts features from the partial sequence. Following the existing work [74, 89], LEAM considers three kinds of heterogeneous information in a partial sequence: (1) *Grammar rule sequence*: LEAM first considers the position of each grammar rule in the partial sequence by representing the partial sequence as a vector based on the ID of each grammar rule. We take it as the input of the above self-attention layer and denote the output as $(r_1, r_2, \ldots, r_P)$, where P is the length of the partial sequence. (2) *Grammar rule semantics*: LEAM considers the semantics of each grammar rule by its definition. It represents the definition of each grammar rule as a vector by character embedding of each symbol in the definition with a fully-connected layer, which is represented as $(v_1, v_2, \ldots, v_P)$. (3) *Grammar rule path*: a partial sequence of grammar rules can construct a partial AST, and thus LEAM further considers the rule path from the root node to the current node to be expanded, which can indicate the depth of each grammar rule in the partial

AST. It represents the rule path as a vector based on the ID of each node, denoted as ($q_1, q_2, \ldots, q_P$). These features can be further processed by gating (ϕ_{gating}) and grammar-attention ($\phi_{grammar}$) layers as shown in Formula 4.

$$\begin{aligned} u_1, u_2, \ldots, u_P &= \phi_{gating}([r_1, r_2, \ldots, r_P; v_1, v_2, \ldots, v_P]) \\ g_1, g_2, \ldots, g_P &= \phi_{grammar}([u_1, u_2, \ldots, u_P; q_1, q_2, \ldots, q_P]) \end{aligned} \tag{4}$$

Then, the final feature vector ($d_1, d_2, \ldots, d_T$) can be computed by the fully-connected layer (ϕ_{dense}) as shown in Formula 5.

$$d_1, d_2, \ldots, d_T = \phi_{dense}([a_1, a_2, \ldots, a_L; g_1, g_2, \ldots, g_P]) \tag{5}$$

3.3 Grammar Rule Prediction

Our DL model is based on the state-of-the-art syntax-guided code generation model, i.e., TreeGen [74]. It is a tree-based Transformer [78] and thus it can solve the significant challenge of long dependencies between code elements [78]. Due to its effectiveness, it has been widely-used in many code-related tasks [24, 74, 84, 89].

In our task, LEAM designs two sub-models in the TreeGen architecture for our two-step prediction (i.e., statement prediction and change prediction). Specifically, the final feature vector ($d_1, d_2, \ldots, d_T$) is fed into the two sub-models for statement prediction and change prediction. Both of them are pointer networks [79], which are trained by maximizing the negative log-likelihood of the ground-truth sequence of grammar rules for each pair of correct and faulty methods in training data. In particular, they do not work at the same time by designing a control module to determine which sub-model is enabled. It first enables the sub-model for statement prediction, in order to identify the statements to be mutated. Then, it enables the sub-model for change prediction and disables the other sub-model, in order to change the identified statements by predicting a sequence of grammar rules. The calculation via the pointer network can be shown as Formulae 6 and 7.

$$\gamma_1, \gamma_2, \ldots, \gamma_T = \phi_{pointer}([d_1, d_2, \ldots, d_T]) \tag{6}$$

$$p_i = \frac{exp(\gamma_i)}{\sum_{j=1}^{T} exp(\gamma_j)} \tag{7}$$

where ($\phi_{pointer}$) represents the pointer network. The output of the pointer network ($\gamma_1, \gamma_2, \ldots, \gamma_T$) is then normalized by softmax to obtain normalized vector ($p_1, p_2, \ldots, p_T$). Specifically, the output of the sub-model for statement prediction is the probability of each statement to be mutated under the context of the targeted method. The output of the sub-model for change prediction is the probability of each grammar rule (except *Rules 1-2*) to be applied to expand the current non-terminal. For the rules whose left side is not the current non-terminal, LEAM sets the outputs of the fully-connected layer to $-\infty$ and then their probabilities can be 0 after softmax normalization.

Please note that, the current LEAM implementation supports the mutation on one or two statements due to the following two reasons: (1) it can effectively reduce the search space; (2) a large percentage of real faults involve at most two statements. We analyzed the number of statements involved in each real fault of our collected data (i.e., 297,029 real faults of Java projects, to be presented in Section 4.1), and found that the real faults involving one and two statements occupy 87.35%. In the future, we can further extend LEAM to support the mutation on more statements.

3.4 Mutation Fault Construction

With our DL model, LEAM constructs a mutation fault by applying a sequence of grammar rules (predicted by change prediction) to the statements (predicted by statement prediction). If LEAM only preserves the most probable sequence of grammar rules on the most probable statements, only one mutation fault can be constructed for the targeted method. However, in practice, a method may introduce different faults, and thus it is important to construct a set of mutation faults for a targeted method. To balance the accuracy and efficiency of mutation fault construction, LEAM incorporates the beam search algorithm [26, 71] to obtain a set of highly possible sequences of grammar rules. Please note that each sequence of grammar rules in the searched set includes the grammar rules for change prediction and the grammar rules for statement prediction, since such a pair can help construct a mutation fault. Specifically, it preserves Top-K (K refers to beam size) partial sequences of grammar rules in each prediction. For each of the K partial sequences, it then produces Top-K grammar rules as the potential next grammar rules in the final set of grammar rule sequences, and thus obtains K^2 partial sequences of grammar rules. It further preserves Top-K partial sequences according to the probabilities of the K^2 partial sequences for next prediction. Following the existing work [43, 44], the probability of a partial sequence of grammar rules is calculated by the product of the probability of each grammar rule in the partial sequence at the corresponding prediction.

When all the non-terminals have been expanded, the prediction process stops and a final set of Top-K sequences of predicted grammar rules are obtained for mutation fault construction. Also, to guarantee the search efficiency, we also terminate the prediction of a sequence of grammar rules when its length reached a pre-defined threshold ζ. That is, we will discard such sequences of grammar rules for mutation fault construction.

4 EVALUATION DESIGN

In this section, we conducted an extensive study to sufficiently evaluate the quality of mutation faults constructed by our proposed mutation technique, i.e., LEAM. Specifically, we evaluated LEAM in the three usage scenarios for mutation faults, including its original scenario (i.e., mutation testing) – **RQ1**, a widely-studied software testing task (i.e., mutation-based TCP) – **RQ2**, and a widely-studied software debugging task (i.e., mutation-based FL) – **RQ3**. Both TCP and FL are important downstream applications of mutation faults.

4.1 Training Data and Subjects Under Test

We constructed the training data required by LEAM based on the open-source dataset provided by the existing work [89]. It contains the Java projects created between March 2011 and March 2018 on GitHub [1], and collected fault-fixing commits by checking whether the commit messages contain at least one of the following words: *bug*, *issue*, *problem*, *error*, *fix*, and *solve*. Same as the existing work [77], we obtained a real fault by treating a fault-fixing commit as the clean version and its prior commit as the faulty version by introducing a fault to the clean version. As presented in Section 3.3, LEAM currently supports the mutation on one or two statements, and thus we further removed the faults involving more than two statements. To avoid data leakage, we removed all the commits related to the projects in Defects4J V1.0 (that is our subjects to be introduced later). Finally, we obtained 297,029 real faults, and used 80% of them as the training set and 20% as the validation set.

To evaluate LEAM, we adopted Defects4J [41], one of the most widely-used benchmarks in the area of software testing and debugging [10, 11, 37, 51, 53, 76, 89], as subjects in our study. Specifically, we used all the five subjects (i.e., Commons **Lang**, Joda-**Time**, Commons **Math**, JFree**Chart**, and **Closure** Compiler) with 357 real faults in Defects4J V1.0. Since mutation testing is very costly and our study needs to run four mutation techniques, we did not use more subjects in the latest Defects4J version in order to balance the study sufficiency and costs. Indeed, Defects4J V1.0 has been the most widely used Defects4J version for existing studies on mutation testing and its downstream applications [10, 19, 37, 38, 42, 52, 53].

4.2 Compared Mutation Techniques

In our study, we compared LEAM with two most widely-used traditional mutation techniques, i.e., **Major** [40] and **PIT** [21], and the state-of-the-art DL-based mutation technique, i.e., **DeepMutation** (DM) [76]. All of them are open-source tools. Major and PIT conduct syntactic mutation on *source code* and *bytecode* respectively, based on their corresponding pre-defined mutation operators. In our study, we used all the mutation operators in them for constructing mutation faults, respectively. The details on DeepMutation have been introduced in Section 1. Regarding it, we used the configurations recommended by its original paper [77]. Both DeepMutation and LEAM conduct source-code-level mutation like Major. Please note that same as the existing work [60], DeepMutation cannot be applied to Closure due to its internal errors, and thus we cannot obtain the results of DeepMutation on Closure. As we compared these techniques in terms of overall results across all the subjects in our paper, our overall results did not include the results on Closure in order to fairly compare with DeepMutation. The results on Closure can be found at our project homepage [2] and the conclusions on Closure are consistent with the overall conclusions in the paper.

4.3 Implementations

We implemented LEAM in Python 3.7.0 based on PyTorch 1.3.0. We determined the settings of the hyper-parameters in LEAM based on the performance on our validation set. Specifically, we set the embedding size to 256, the size of hidden layers to 256, the optimizer to Adam, the learning rate to 0.0001, and the number of epochs to 20. We set the beam size to 64, and ζ to 60. To promote the practical use and future research, we have released both the implementation of LEAM and our built model at our project homepage [2]. With our implementation, researchers/practitioners can replicate our experiments, improve the performance of LEAM in future research, and extend LEAM to other programming languages. With our built model, users can save the training time and directly use it to construct mutation faults for any given Java projects.

We conducted all the experiments on a server with Intel(R) Xeon(R) Silver 4214 @ 2.20GHz CPU, 256GB memory, NVIDIA GeForce RTX 2080 Ti, and Ubuntu 18.04 as the operating system.

5 RESULTS AND ANALYSIS

5.1 RQ1: Effectiveness Comparison in Mutation Testing

5.1.1 Metrics. We first compared the quality of mutation faults constructed by the four mutation techniques in the original scenario, i.e., mutation testing. We adopted a set of widely-used metrics in mutation testing to measure the quality of mutation faults in this scenario [4, 42, 60, 85]. The first metric measures how the mutation faults can represent real faults in terms of *adequate* test suites [60, 85]; as a supplement to the first metric, the second metric further measures how the mutation faults can represent the mutation faults constructed by other mutation techniques [49]. Furthermore, the last metric measures how mutation faults can represent real faults in terms of *non-adequate* test suites [42].

Regarding the first metric, following the existing work [4, 42, 85], for each mutation technique, we constructed the minimal test suite that is selected from the original test suite but kills the maximum number of mutation faults constructed by this technique, and then measured the percentage of real faults killed by the constructed test suite. Regarding the second metric, following the existing work [4, 42], for each mutation technique, we further measured the mutation score of the above-constructed test suite on the mutation faults constructed by each of the other mutation techniques, respectively. Regarding the last metric, similar to the existing work [41], for each mutation technique, we constructed m test suites, each of which contains n test cases selected from the original test suite randomly without replacement, and then measured the mutation score and the result of real fault detection for each test suite. Finally, we measured the point-biserial correlation (that measures the correlation between dichotomous variable and continuous variable) [75] between mutation score and real fault detection. Please note that we removed the mutation faults that cannot be killed by the original test suite for the above metrics following the existing work [33].

5.1.2 Process. Following the existing work [42], we constructed mutation faults by each studied mutation technique on each fixed version in each subject, and regarded the corresponding faulty version as the real fault introduced to the fixed version. Specifically, we considered only the changed source files between the fixed version and the faulty version for constructing mutation faults. Then, we measured the quality of the mutation faults constructed by each mutation technique in terms of the three metrics. Please note that for each metric, we repeated the process 10 times to reduce the influence of randomness. For the last metric, we set $m = 50$ and $n = 50$ similar to the existing study [87]. If the number of test cases in the original test suite cannot support the construction of 50 test suites, we constructed the maximum number of test suites with the size of 50. If it cannot support the construction of 10 test suites with the size of 50, we discarded this version since it does not have statistical significance for the correlation.

Besides, different mutation techniques tend to construct different numbers of mutation faults, which could affect the effectiveness of each mutation technique in terms of these metrics. Although the number of constructed mutation faults is the inherent characteristic of each mutation technique, we still tried to compare them by controlling for the number of mutation faults. On average, the number of mutation faults constructed by DeepMutation for each changed source file (i.e., 11) is the smallest among the four techniques since (1) it cannot ensure the syntactic correctness of each constructed mutation fault; and (2) it is just applicable to the methods where the number of tokens is smaller than 50 in order to improve the learning process. The number of mutation faults constructed by PIT (i.e., 2,719) is the largest since it conducts mutation on bytecode rather than source code. The average number of mutation faults constructed by Major and LEAM is 597 and 347, respectively.

Overall, we first compared the four techniques in terms of the three metrics when using all the constructed mutation faults. Then, we compared the four techniques by randomly sampling the same number of mutation faults for each mutation technique as the smallest number of constructed mutation faults among the four techniques on each version of each subject. Finally, since the number of constructed mutation faults by DeepMutation is largely smaller than those by the other three techniques due to the above-mentioned reasons, keeping the number of mutation faults same as that of DeepMutation may incur bias. We further repeated the experiment by leaving DeepMutation out and randomly sampling the same number of mutation faults as the smallest number of constructed mutation faults among the three techniques (i.e., Major, PIT, and LEAM) for each version of each subject. To reduce the influence of randomness caused by mutation fault sampling, we repeated the experiments 10 times.

5.1.3 Results. Figure 2 shows the quality of the mutation faults constructed by each mutation technique in terms of the first metric. Each figure shows the percentage of real faults killed by the constructed test suite based on the mutation faults for each mutation technique. We put the results on all the versions of all the subjects together to draw each box. From Figure 2, LEAM detects the largest percentage of real faults by the constructed test suite based on its constructed mutation faults among all the studied techniques, regardless of using all the constructed mutation faults or controlling for the number of mutation faults. For example, when controlling for the number of mutation faults same as the smallest one among Major, PIT, and LEAM, the medium percentage of real faults detected by LEAM across all the versions is 75.35%, while that by Major and PIT are 55.46% and 35.29%, respectively. The results demonstrate that LEAM is more helpful to construct mutation faults representing real faults in terms of adequate test suites.

Similarly, Figure 3 shows the quality of the mutation faults constructed by each mutation technique in terms of the second metric. We take the first group of boxes in Figure 3(a) as the representative to explain how to read this kind of figures. Each box in this group shows the mutation score of the test suite constructed based on the mutation faults constructed by Major, in killing the mutation faults constructed by PIT, DeepMutation, and LEAM, respectively. For ease of presentation, we call them the mutation scores of Major over PIT, Major over DeepMutation, and Major over LEAM, respectively. From Figure 3(a), the median mutation score of Major over LEAM, PIT over LEAM, and DeepMutation over LEAM are 0.90, 0.91, and 0.38 respectively, while that of LEAM over Major, LEAM over PIT, and LEAM over DeepMutation are 0.99, 0.98, and 0.99 respectively. The results demonstrate that the mutation faults constructed by LEAM can better represent the mutation faults constructed by other techniques when using all the constructed mutation faults. The same conclusion can be obtained when controlling for the number of constructed mutation faults from Figures 3(b) and 3(c).

Table 2: Correlation of mutation score and real fault detection

Tech.	Mutation Faults Number		
	All	Control w/ DM	Control w/o DM
Major	0.60	0.30	0.49
PIT	0.56	0.28	0.49
DM	0.28	0.14	-
LEAM	**0.64**	**0.40**	**0.55**

Table 2 shows the medium correlation coefficients between mutation score and real fault detection (i.e., the third metric) across all the versions of all the subjects. The second column shows the result when using all the constructed mutation faults, while the last two columns show the results under the two scenarios of controlling for the number of mutation faults. From Table 2, the correlation coefficient of LEAM is larger than that of the other three techniques regardless of using all the mutation faults or controlling for the number of mutation faults. Moreover, all the p-values for these correlations are smaller than 0.05, indicating that the correlations have statistical significance. Hence, the results demonstrate that there is stronger correlation between mutation score on mutation faults constructed by LEAM and real fault detection, compared with the other three techniques.

5.2 RQ2: Effectiveness Comparison in Mutation-based Test Case Prioritization

5.2.1 Metrics. In the scenario of mutation-based TCP, we measured the effectiveness of each studied mutation-based TCP technique by feeding the mutation faults constructed by a mutation technique, as the metric for the quality of the mutation faults constructed by this mutation technique. Following the existing TCP work [25, 55, 70], we adopted the most widely-used metric, i.e., **APFD** (Average Percentage of Faults Detected) [68], to measure the effectiveness of a mutation-based TCP technique: $APFD = 1 - \frac{TF_1+TF_2+...+TF_r}{nr} + \frac{1}{2n}$, where r refers to the number of faults to be detected by the test suite to be prioritized, n is the number of test cases in the test suite, TF_i is the ranking of the first test case in the test suite prioritized by a mutation-based TCP technique that detects the i^{th} fault.

For a mutation-based TCP technique, if using the mutation faults constructed by a mutation technique achieves the larger APFD value than that by another mutation technique, it means that the former mutation technique outperforms the latter.

5.2.2 Process. Following the existing study [70], for each pair of versions in each subject, we mimicked the regression testing scenario by treating the fixed version as the prior version and the faulty version as the current version under test. Then, we constructed mutation faults on the previous version via each mutation technique. Here, we considered only the changed source files between the two versions for constructing mutation faults following the existing study [70]. Based on the mutation faults constructed by a mutation

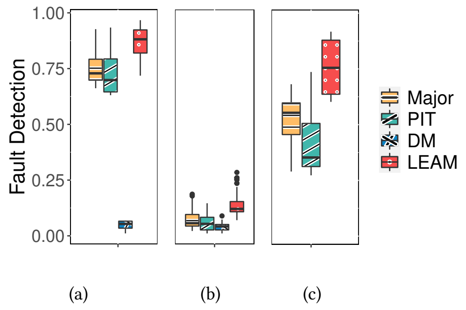

Figure 2: Ability of representing real faults in terms of adequate test suites. (a) shows the result when using all the constructed mutation faults; (b)/(c) shows the result under controlling for the number of mutation faults when considering/ignoring DeepMutation (DM).

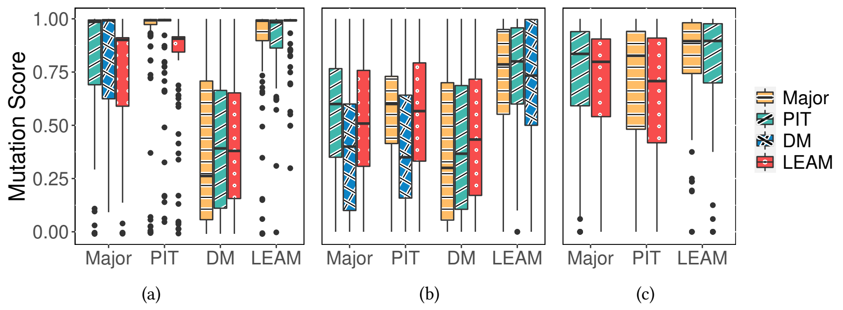

Figure 3: Ability of representing mutation faults constructed by other mutation techniques. (a) shows the result when using all the constructed mutation faults; (b)/(c) shows the result under controlling for the number of mutation faults when considering/ignoring DeepMutation (DM).

Table 3: Overall effectiveness in mutation-based TCP

TCP	Major	PIT	DM	LEAM
GRK	0.62	0.65	0.20	**0.75**
GRD	0.62	0.65	0.20	**0.76**
HYB-ω	0.61	0.65	0.20	**0.75**

technique, we prioritized test cases in the test suite provided by the previous version via each studied mutation-based TCP technique (i.e., GRK, GRD, and HYB-ω), and then calculated the APFD value for the prioritized test suite on the corresponding faulty version.

In this scenario (and the scenario of mutation-based FL), we did not control for the number of mutation faults, but used all the mutation faults constructed by each mutation technique. The reasons are twofold: (1) the number of constructed mutation faults is essentially the inherent characteristic of a mutation technique; (2) we have demonstrated the effectiveness of LEAM in the scenario of mutation testing regardless of using all the constructed mutation faults or controlling for the number of mutation faults in Section 5.1.3. More discussion about the influence of the number of constructed mutation faults can be found in Section 5.2.3.

5.2.3 Results. Table 3 shows the average APFD results across all the versions of all the subjects. We found that all the studied mutation-based TCP techniques achieve better TCP effectiveness by feeding the mutation faults constructed by LEAM than feeding those by the three compared techniques on average. The same conclusion can be obtained on each subject, but due to space limit, we put the detailed results on each subject to our project homepage [2]. Specifically, LEAM achieves 20.97%, 15.38%, and 275.00% improvements over Major, PIT, and DeepMutation in terms of average APFD across all the subjects for GRK, 22.58%, 16.92%, and 280.00% improvements for GRD, and 22.95%, 15.38%, and 275.00% improvements for HYB-ω. Furthermore, we performed *a paired sample Wilcoxon signed-rank test* [82] at the significance level of 0.05 to investigate whether LEAM significantly outperforms each compared technique on each subject for each studied mutation-based TCP technique. All the calculated p-values are smaller than 1.86e-5 (that is far smaller than 0.05), and thus the results demonstrate the statistically significant superiority of LEAM over all the compared mutation techniques. Overall, LEAM significantly outperforms all the compared mutation techniques in the scenario of mutation-based TCP.

From our results, LEAM performs significantly better regardless of comparison with PIT that constructs the largest number of mutation faults on average or comparison with DeepMutation that constructs the smallest number of mutation faults on average in this scenario (as well as the scenario of mutation-based FL to be presented in Section 5.3.3). The results further confirm the stable effectiveness of LEAM independent of the number of mutation faults to some degree. Also, we investigated the influence of the number of mutation faults, which can be controlled by *beam size* (a hyper parameter in LEAM), on LEAM. Due to space limit, we put the detailed results to our project homepage [2], and the conclusion is that with the beam size increasing, the effectiveness of LEAM in both mutation-based TCP and mutation-based FL becomes better. By balancing the effectiveness and efficiency, our default setting of beam size (i.e., 64) is a good choice.

5.3 RQ3: Effectiveness Comparison in Mutation-based Fault Localization

5.3.1 Metrics. In the scenario of mutation-based FL, we measured the effectiveness of each studied mutation-based FL technique by feeding the mutation faults constructed by a mutation technique, as the metric for the quality of the mutation faults constructed by this mutation technique. Following the existing FL work [51, 52], we adopted three most widely-used metrics to measure the effectiveness of a mutation-based FL technique on each subject. (1) **Top-N**: the number of successfully localized faults within the Top-N position in the ranking list produced by a FL technique. Following the existing studies [51, 52], we considered N to be 1, 3, 5, respectively. (2) Mean First Rank (**MFR**): the mean of the first faulty method rank for each fault. This metric emphasizes

Table 4: Overall effectiveness in mutation-based FL

FL	Tech.	Top-1	Top-3	Top-5	MFR	MAR
Metallaxis	Major	35	92	114	9.56	12.42
	PIT	56	102	128	8.16	11.83
	DM	19	47	98	16.64	20.65
	LEAM	**118**	**182**	**188**	**3.86**	**4.57**
MUSE	Major	35	89	111	10.99	13.11
	PIT	52	97	124	9.15	11.72
	DM	18	53	94	18.70	22.47
	LEAM	**126**	**181**	**189**	**3.88**	**5.05**

fast localization of the first faulty element to ease debugging. (3) Mean Average Rank (**MAR**): the mean of the average rank of all the faulty methods for each fault. Different from MFR, MAR emphasizes precise localization for all the faulty elements. If more than two methods have the same suspicious scores, we used the $E_{inspect}$ [90] to calculate the expected rank following the existing work [90].

For a mutation-based FL technique, if using the mutation faults constructed by a mutation technique achieves the larger Top-N value, smaller MFR value, or smaller MAR value than that by another technique, it means that the former outperforms the latter.

5.3.2 Process. Following the existing studies [51, 52], for each faulty version in each subject, we first constructed mutation faults by each mutation technique respectively, and then used each studied mutation-based FL technique to produce a ranking list of suspicious methods based on the mutation faults constructed by each mutation technique respectively. Finally, we measured the above three metrics on each subject based on the corresponding produced ranking list of suspicious methods.

5.3.3 Results. Table 4 shows the overall effectiveness of each studied mutation-based FL technique based on the mutation faults constructed by each mutation technique across all the subjects. We found that for both Metallaxis and Muse, LEAM achieves better FL effectiveness than all the compared techniques in terms of all the metrics. The same conclusion can be obtained on each subject, but due to space limit, we put the detailed results on each subject to our project homepage [2]. Overall, for Metallaxis, LEAM achieves 237.14%, 110.71%, 521.05% improvements over Major, PIT, DeepMutation in terms of Top-1, 59.62%, 52.70%, 76.80% improvements in terms of MFR, 63.20%, 61.37%, 77.87% improvements in terms of MAR. For MUSE, LEAM achieves 260.00%, 142.31%, 600.00% improvements over Major, PIT, DeepMutation in terms of Top-1, 64.70%, 57.60%, 79.25% improvements in terms of MFR, 61.48%, 56.91%, 77.53% improvements in terms of MAR. We also performed a *paired sample Wilcoxon signed-rank test* [82] at the significance level of 0.05 to investigate whether LEAM significantly outperforms each compared technique on each subject for each studied mutation-based FL technique. All the p-values are smaller than 3.52e-5 (that is far smaller than 0.05), and thus the results demonstrate the statistically significant superiority of LEAM over all the compared mutation techniques. Therefore, LEAM significantly outperforms all the compared mutation techniques in the scenario of mutation-based FL.

Table 5: Comparison effectiveness between LEAM and its variant w/o statement prediction (SP) in mutation-based TCP

TCP	GRK	GRD	HYB-ω
LEAM w/o SP	0.60	0.60	0.59
LEAM	0.75	0.76	0.75

6 DISCUSSION

6.1 Contribution of Statement Prediction

One major component in LEAM is statement prediction, which aims to identify the statements highly possible to be mutated under the context of the targeted method and thus largely reduces the search space for better prediction. Therefore, it is necessary to investigate whether this component really contributes to LEAM. Here, we conducted an experiment by comparing LEAM with its variant removing the component of statement prediction in the scenarios of mutation-based TCP and mutation-based FL. That is, this variant randomly selects at most two statements for mutation fault construction in LEAM. Due to space limit, we just reported the average results across all the subjects in mutation-based TCP (in Table 5), and the results in mutation-based FL can be found at our project homepage [2]. We found that LEAM largely outperforms the variant of LEAM without the statement prediction component in terms of average APFD for each studied mutation-based TCP technique. The results confirm the significant contribution of the statement prediction component in LEAM.

6.2 Efficiency of LEAM

Although the time spent on our experiments mainly lies in executing the constructed mutation faults with test cases, it is also important to investigate the time spent on mutation fault construction in order to understand the efficiency of each mutation technique. Since each mutation technique may construct different numbers of mutation faults, we calculated the average time spent on constructing a mutation fault to fairly measure the efficiency of each mutation technique. Specifically, the average time for Major, PIT, DeepMutation, and LEAM is 0.04s, 0.003s, 0.70s, and 0.64s, respectively. The results show that all of these mutation techniques are efficient in terms of the time spent on mutation fault construction, and DL-based techniques spent longer time than traditional techniques as expected.

Besides, DL-based mutation techniques involve the training process. The training time for LEAM is about 24 hours, and we cannot report the training time for DeepMutation since we directly used its pre-trained model. Since the training process is offline and the built model can be directly used by users for mutation without retraining, the training time of LEAM is actually acceptable in practice. Overall, all the mutation techniques have high efficiency in mutation fault construction, and LEAM also has acceptable training time, which demonstrates the practicability of LEAM by comprehensively considering its effectiveness and efficiency.

6.3 Threats to Validity

The threat to *internal* validity mainly lies in our implementation for LEAM. To reduce this kind of threat, we implemented LEAM based on mature libraries described in Section 4.3, and two authors have carefully checked our code.

The threat to *external* validity mainly lies in the subjects used in our study. Although LEAM is general, we just evaluated its effectiveness on Java projects (the widely-used Defects4J benchmark [41]), which may not represent the subjects under other programming languages. In the future, we will extend and evaluate LEAM on more diverse subjects with different programming languages, in order to further reduce this kind of threat.

The threat to *construct* validity mainly lies in the configurations of LEAM. To reduce this kind of threat, we have reported the configurations of LEAM in our experiments in Section 4.3 and investigated the influence of the important hyper-parameter (i.e., beam size) as presented in Section 5.2.3. In the future, we will further investigate the influence of other hyper-parameters in LEAM in order to further reduce this kind of threat.

7 RELATED WORK

In the literature, many mutation techniques have been proposed in the area of mutation testing [21, 30, 35, 36, 69]. Most of them require developers to design mutation operators, each of which can conduct a simple syntactic change to the program under test, for constructing mutation faults. Typical mutation techniques in this category include MuJava [56], Javalanche [69], Major [40] and PIT [21], etc. As demonstrated by the existing studies [12, 28], the mutation faults constructed by traditional mutation techniques may not represent real faults very well. To improve the quality of mutation faults, Brown et al. [12] suggested to extract mutation patterns from real-world bug-fixes to create mutation faults (called wild-caught mutants). Inspired by this idea, DeepMutation [77], the state-of-the-art DL-based technique, was proposed for constructing mutation faults by learning from a large number of real faults via classic sequence-to-sequence NMT. In our study, we chose the more advanced DeepMutation as a compared technique. More details about DeepMutation have been discussed in Section 1. In contrast, LEAM aims to further resolve the limitations of DeepMutation via the syntax-guided encode-decoder architecture, and has been shown to substantially outperform DeepMutation. Our experiments have demonstrated the effectiveness of LEAM compared with two typical traditional techniques and DeepMutation.

Patra and Pradel [65] proposed SemSeed, which extracts mutation patterns from real-world bug-fixes and then generalizes them to other code locations by measuring the similarities of identifiers and literals based on learned token embeddings. Actually, it is not a general-purpose mutation technique, since it (1) simply reverts a bug-fixing code change to a mutation pattern, (2) supports only one-line pattern, and (3) just considers the semantics of identifiers and literals but ignores the program/method semantics. In particular, the tool is specific to JavaScript programs. Therefore, same as the existing study [60], we did not compare with SemSeed (our study is based on Java programs) due to its totally different design and targeted programming language [65].

Also, Khanfir et al. [46] proposed IBIR, which utilizes natural languages in *bug reports* to decide the mutation locations and then applies the mutation patterns from a pattern-based program repair tool (i.e.,TBar [54]) at the identified locations. Beller et al. [9] proposed *Mutation Monkey*, which is a *semi-automatic* technique by mining patterns from historical changes and then transforming these patterns to mutation operators. Different from them, LEAM uses the syntax-guided encoder-decoder architecture to automatically construct mutation faults by learning from a large number of real faults at the AST level.

Lastly, there are many empirical studies on investigating the effectiveness of existing mutation techniques in the literature [23, 47, 48, 60]. They tend to evaluate the quality of constructed mutation faults by the studied mutation techniques in terms of metrics in mutation testing. Different from them, we not only adopted the widely-used metrics in mutation testing, but also investigated the quality of constructed mutation faults in two downstream applications of mutation faults (i.e., mutation-based TCP and FL).

8 CONCLUSION

In this work, we propose a novel DL-based technique (i.e., LEAM) to construct mutation faults by learning from real faults. It adapts the syntax-guided encoder-decoder architecture by extending a set of grammar rules specific to our mutation task, in order to ensure syntactic correctness of constructed mutation faults. Moreover, it significantly reduces search space by first predicting the statements to be mutated in a targeted method and improves model performance by extracting more comprehensive features from AST. Our extensive study on Defects4J demonstrates the effectiveness of LEAM in three popular scenarios (including mutation testing, mutation-based TCP, and mutation-based FL) compared with two traditional techniques and the state-of-the-art DL-based technique.

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China Grant No. 62002256, and National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474.

REFERENCES

[1] Accessed: 2022. GitHub. https://github.com/.
[2] Accessed: 2022. LEAM. https://github.com/tianzhaotju/LEAM.
[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.
[4] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. 2014. Establishing theoretical minimal sets of mutants. In *2014 IEEE seventh international conference on software testing, verification and validation*. IEEE, 21–30.
[5] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th international conference on Software engineering*. 402–411.
[6] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.
[7] Fatmah Yousef Assiri and James M Bieman. 2017. Fault localization for automated program repair: effectiveness, performance, repair correctness. *Software Quality Journal* 25, 1 (2017), 171–199.
[8] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 177–188.
[9] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry—A Study at Facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 268–277.
[10] 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 *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 907–918.

[11] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2021. Evaluating and Improving Unified Debugging. *IEEE Transactions on Software Engineering* (2021).

[12] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 511–522.

[13] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering*. IEEE, 700–711.

[14] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. Test case prioritization for compilers: A text-vector based approach. In *2016 IEEE international conference on software testing, verification and validation*. IEEE, 266–277.

[15] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 223–234.

[16] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 656–667.

[17] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 78–89.

[18] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2018. Coverage prediction for accelerating compiler testing. *IEEE Transactions on Software Engineering* 47, 2 (2018), 261–278.

[19] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and Precise On-the-fly Patch Validation for All. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1123–1134.

[20] Lingchao Chen and Lingming Zhang. 2018. Speeding up mutation testing via regression test selection: An extensive study. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 58–69.

[21] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis*. 449–452.

[22] Murial Daran and Pascale Thévenod-Fosse. 1996. Software error analysis: A real case study involving real faults and mutations. *ACM SIGSOFT Software Engineering Notes* 21, 3 (1996), 158–171.

[23] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John Clark, and Inmaculada Medina-Bulo. 2018. Evaluation of mutation testing in a nuclear industry case study. *IEEE Transactions on Reliability* 67, 4 (2018), 1406–1419.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[25] Hyunsook Do and Gregg Rothermel. 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 32, 9 (2006), 733–752.

[26] Markus Freitag and Yaser Al-Onaizan. 2017. Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806* (2017).

[27] 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*. 19–30.

[28] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How close are they to real faults?. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 189–200.

[29] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. 2014. A unified test case prioritization approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–31.

[30] Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood, and Darko Marinov. 2019. Comparing mutation testing at the levels of source code and compiler intermediate representation. In *2019 12th IEEE Conference on Software Testing, Validation and Verification*. IEEE, 114–124.

[31] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing white-box and black-box test prioritization. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 523–534.

[32] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-based fault localization for real-world multilingual programs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 464–475.

[33] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*. 435–445.

[34] Sean A Irvine, Tin Pavlinic, Leonard Trigg, John G Cleary, Stuart Inglis, and Mark Utting. 2007. Jumble java byte code to measure the effectiveness of unit tests. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 169–175.

[35] Yue Jia and Mark Harman. 2009. Higher order mutation testing. *Information and Software Technology* 51, 10 (2009), 1379–1393.

[36] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[37] Jiajun Jiang, Yingfei Xiong, and Xin Xia. 2019. A manual inspection of defects4j bugs and its implications for automatic program repair. *Science China Information Sciences* 62, 10 (2019), 1–16.

[38] 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*. 298–309.

[39] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 467–477.

[40] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 international symposium on software testing and analysis*. 433–436.

[41] 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 2014 International Symposium on Software Testing and Analysis*. 437–440.

[42] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.

[43] Yuning Kang, Zan Wang, Hongyu Zhang, Junjie Chen, and Hanmo You. 2021. APIRecX: Cross-Library API Recommendation via Pre-Trained Language Model. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 3425–3436.

[44] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1073–1085.

[45] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 114–125.

[46] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2020. Ibir: Bug report driven fault injection. *arXiv preprint arXiv:2012.06506* (2020).

[47] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, and Nicos Malevris. 2016. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 147–156.

[48] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. 2018. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering* 23, 4 (2018), 2426–2463.

[49] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. 2017. Assessing and improving the mutation testing practice of pit. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 430–435.

[50] Tien-Duy B Le, Richard J Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 579–590.

[51] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 169–180.

[52] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[53] Jingjing Liang, Ruyi Ji, Jiajun Jiang, Shurui Zhou, Yiling Lou, Yingfei Xiong, and Gang Huang. 2021. Interactive Patch Filtering as Debugging Aid. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 239–250.

[54] 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*. 31–42.

[55] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 46–57.

[56] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability* 15, 2 (2005), 97–133.

[57] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.

[58] Ivan Moore. 2001. Jester-a JUnit test tester. *Proc. of 2nd XP* (2001), 84–87.

[59] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[60] Milos Ojdanic, Aayush Garg, Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2021. Syntactic Vs. Semantic similarity of Artificial and Real Faults in Mutation Testing Studies. *arXiv preprint arXiv:2112.14508* (2021).

[61] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 354–365.

[62] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[63] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[64] David Paterson, Gregory M Kapfhammer, Gordon Fraser, and Phil McMinn. 2018. Using controlled numbers of real faults and mutants to empirically evaluate coverage-based test case prioritization. In *Proceedings of the 13th International Workshop on Automation of Software Test*. 57–63.

[65] Jibesh Patra and Michael Pradel. 2021. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 906–918.

[66] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.

[67] Manos Renieres and Steven P Reiss. 2003. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*. IEEE, 30–39.

[68] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 179–188.

[69] David Schuler and Andreas Zeller. 2009. Javalanche: Efficient mutation testing for Java. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 297–298.

[70] Donghwan Shin, Shin Yoo, Mike Papadakis, and Doo-Hwan Bae. 2019. Empirical evaluation of mutation-based test case prioritization techniques. *Software Testing, Verification and Reliability* 29, 1-2 (2019), e1695.

[71] Raphael Shu and Hideki Nakayama. 2018. Improving beam search by removing monotonic constraint for neural machine translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 339–344.

[72] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 273–283.

[73] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 7055–7062.

[74] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[75] Robert F Tate. 1954. Correlation between a discrete and a continuous variable. Point-biserial correlation. *The Annals of mathematical statistics* 25, 3 (1954), 603–607.

[76] Michele Tufano, Jason Kimko, Shiya Wang, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, and Denys Poshyvanyk. 2020. DeepMutation: a neural mutation tool. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 29–33.

[77] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 301–312.

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

[79] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. *Advances in neural information processing systems* 28 (2015).

[80] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing test inputs for deep neural networks via mutation analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 397–409.

[81] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2348–2368.

[82] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968

[83] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-shot Learning. In *FSE*.

[84] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems* 32 (2019).

[85] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. 2013. Operator-based and random mutant selection: Better together. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 92–102.

[86] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2012. Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 331–341.

[87] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 214–224.

[88] Jianyi Zhou, Junjie Chen, and Dan Hao. 2021. Parallel test prioritization. *ACM Transactions on Software Engineering and Methodology* 31, 1 (2021), 1–50.

[89] 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.

[90] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* 47, 2 (2019), 332–347.