This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

Generating Python Mutants from Bug Fixes using Neural Machine Translation

SERGEN AŞIK[1,2,3], UĞUR YAYAN[1,2,3]

[1]Department of Software Engineering, Eskisehir Osmangazi University, Eskisehir 26040, Turkey
[2]Autonomous Systems & Reliability Laboratory, Eskisehir Osmangazi University, Eskisehir 26040, Turkey
[3]Intelligent Factory and Robotics Laboratory, Eskisehir Osmangazi University, Eskisehir 26040, Turkey

Corresponding author: Sergen Aşık (sergen.asik@ogu.edu.tr)

This work was supported by the VALU3S project, funded by the ECSEL Joint Undertaking (JU) under Grant Agreement No. 876852, and the Scientific and Technical Research Council of Turkey (TUBITAK) under Contract No. 120N800. The project is titled "Verification and Validation of Automated Systems' Safety and Security."

ABSTRACT Due to the fast-paced development of technology, the software has become a crucial aspect of modern life, facilitating the operation and management of hardware devices. Nevertheless, using substandard software can result in severe complications for users, putting human lives at risk. This underscores the significance of error-free and premium-quality software. Verification and validation are essential in ensuring high-quality software development; software testing is integral to this process. Although code coverage is a prevalent method for assessing the efficacy of test suites, it has some limitations. Therefore, mutation testing is proposed as a remedy to tackle these limitations. Furthermore, mutation testing is recognized as a method for directing test case creation and evaluating the effectiveness of test suites. Our proposed method involves autonomously learning mutations from faults in real-world software applications. Firstly, our approach involves extracting bug fixes at the method-level, classifying them according to mutation types, and performing code abstraction. Subsequently, the approach utilizes a deep learning technique based on neural machine translation to develop mutation models. Our method has been trained and assessed using approximately ~588k bug fix commits extracted from GitHub. The results of our experimental assessment indicate that our models can forecast mutations resembling resolved bugs in 6% to 35% of instances. The models effectively revert fixed code to its original buggy version, reproducing the original bug and generating various other buggy codes with up to 94% accuracy. More than 96% of the generated mutants also demonstrate lexical and syntactic accuracy.

INDEX TERMS Software quality, software verification and validation, software testing, mutation testing, mining software repositories, bug fixes, abstract syntax tree, deep learning, neural machine translation, evaluation of generated code quality.

I. INTRODUCTION

The rapid growth of technological advancements has made software an essential component of today's world [1]. Software programs enable computers and other devices, such as smartphones, tablets, smart homes, cars, airplanes, autonomous cars, robots, etc., to operate and control their hardware, allowing them to perform specific tasks [2]–[6]. However, faulty and low-quality software may cause severe problems for users and risk human lives, necessitating the development of error-free and high-quality software products. High-quality software is designed, coded, tested, and documented to meet users' needs at its best and features update, maintenance, and support processes [7]. By minimizing software errors and increasing reliability, these processes contribute to the overall quality of the software. Consequently, verification and validation play a critical role in developing high-quality software [8]. Software testing is part of the verification and validation process to ensure the software meets the desired quality [9]. Software tests verify that the features and functions of the software work correctly, perform well, and are secure. Testing is also done to ensure that the software is error-free [10]. There are various methods in the industry to measure the effectiveness of test suites in detecting faults in software [11]. For example, code coverage is a widely used approach at Google and Facebook [12], [13]. This approach counts

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

how much of a program a test suite covers. Unfortunately, it only satisfies a portion of the targets set by application developers. Moreover, code coverage can be easily cheated since it only determines whether the code is executed [14]. Such disadvantages can be overcome with mutation testing. Mutation testing is a way to assess a software program's quality and the test suite's effectiveness [15]. In mutation testing, a series of deliberate changes (mutations) are applied to the tested parts and scenarios of the software to ensure that all errors are captured [16]. These small and simple changes can alter the behavior of the code. These changes in the code are called mutants. Mutation testing aims to estimate the ability of the software testing process to detect such faults. The mutation score is a metric used in mutation testing to evaluate the software's quality. This score indicates how many of the mutants in the tested software are caught by the tests. A high mutation score suggests that the software is high-quality and has extensive test coverage [17]. A low mutation score, on the other hand, points out that the software is low-quality and has limited test coverage. This measurement provides a way to expand test coverage and improve testing processes [18]. Various static mutation testing tools are available to perform these measurements. For instance, MutPy is an open-source static mutation testing tool for Python [19]. It measures code endurance by creating various mutants in the code. In addition, static mutation testing tools test for potential errors in the program and untested areas by creating copies of the modified code. However, limitations in test coverage, a bias towards producing false-positive results, and the inability to observe the program's runtime behavior are disadvantages of these tools. The Research and Innovation Agenda (SRIA) roadmap document highlights the need to use artificial intelligence technologies to learn fault injection parameters and prioritize testing, aiming to increase the security and resilience of systems in dynamic environments [20]. Neural machine translation (NMT) is an artificial intelligence technology that enables the automatic translation of a text written in one language into another [21]. This technology is based on machine learning and deep learning techniques. NMT models consist of an encoder and a decoder. The encoder layer of the model tries to encode the flexible-length input source sequence into a fixed-length representation called the "context" that captures the meaning of the input sentence. The context is then decoded through the decoder layer and converted to the target sentence. Training is performed on large-scale datasets that connect the source and target languages. Ultimately, NMT-based models learn to do accurate translations using these connections.

This study proposes a mutation testing approach based on NMT using a real-world dataset of bug fixes for the Python programming language over the last ten years. Our contributions to the literature are as follows: (i) We created a dataset of code snippets representing bug fixes and extracted method-level snippets from Python code. (ii) We identified three mutation types for introducing bugs into the code. (iii) We developed an approach to abstracting Python code, which reduces the vocabulary size for an infinite number of identifiers and constants. (iv) We generated mutants using the NMT-based Transformer model, making us the first to use this approach for mutation testing in Python. These mutants are modified code snippets designed to introduce bugs into the code. (v) We evaluated the effectiveness of our approach by comparing the performance of the original code and the mutant code using a set of test cases.

In this paper, we present the following contributions:

- A novel method for deriving code mutants from bug fixes specific to the Python programming language.
- A new dataset has been provided for researchers conducting similar studies.
- A tool for reducing the size of vocabulary in Python has been developed.
- A detailed literature survey has been conducted on the role of NMT in mutation testing.

The rest of the paper is organized as follows: In the Introduction, we highlight the importance of this topic and explain the basic idea. Section II discusses related work in two domains: traditional mutation testing approaches and the application of NMT in software engineering and mutation testing. Section III presents the dataset and model used in our study. Section IV describes the experimental settings for the given dataset and model. Section V details the evaluation design and the results of our approach. We discuss the shortcomings of our study and propose solutions in Section VI. Finally, the conclusion is presented in Section VII.

II. RELATED WORK

A. MUTATION TESTING

Mutation testing, which involves adding minor faults to programs and measuring the effectiveness of test suites in detecting them, was first proposed by DeMillo et al. [22]. Mutation testing operators deliberately change different parts of the code to provide comprehensive software testing. Mutation testing methods define these operators. Method-level mutation testing is a type of mutation testing that focuses on a specific method in the software code to detect faulty behavior. On the other hand, higher-order mutation testing targets a more significant portion of the software, with mutations applied to all code within a class, library, or project. Object-oriented mutation testing is a type of mutation testing designed to test software that utilizes object-oriented programming (OOP) techniques. It tests the implementation of OOP features such as inheritance, multiple inheritance, interfaces, and polymorphism. Finally, concurrency-related mutation testing is designed to test software that involves concurrency. This mutation testing applies synchronization techniques and parallel programming features to detect concurrency errors. Several mutation

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

testing tools implement the mentioned types of mutation testing and operators in some programming languages [23].

Mutation testing tools are available for various programming languages, including C/C++ and Java, which implement many mutation operators. For C/C++ programming languages, these tools implement method-level mutation operators [24]–[30], higher-order mutation operators [31]–[33], object-oriented mutation operators [34], and mutation operators related to concurrency [32], [33]. Similarly, for the Java programming language, mutation testing tools utilize various types of operators such as method-level mutation operators [35]–[40], higher-order mutation operators [35], [36], [41], [42], object-oriented mutation operators [39], [42]. Mutation operators related to concurrency [42]–[44]. Moreover, some tools specifically employ mutation operators designed for the Python programming language [19], [45], [46].

Mutation testing tools have drawbacks, such as limited test coverage, a bias for false-positive results, and an inability to monitor the program's runtime behavior. Therefore, the SRIA roadmap document emphasizes using artificial intelligence technologies to ascertain fault injection parameters and prioritize testing to improve system security and endurance in dynamic environments.

Degiovanni and Papadakis have introduced a mutation testing approach that utilizes the pre-trained CodeBERT language model to generate mutants. This approach works by masking a token within the input code and allowing CodeBERT to predict it. Then, mutants are created by replacing the masked tokens with the predicted ones [47]. One potential limitation of this study could be that the evaluation metrics may need to represent the actual testing cost and effectiveness accurately. Tufano et al. presented a machine-learning method for creating new mutation operators for mutation testing by utilizing the knowledge of how software bugs are fixed. First, the researchers analyzed Java programming language examples to identify the possible reasons for mutation operators and created a dataset based on them. Then, they used the NMT technique to generate mutation operators from this dataset. The authors showed that the proposed approach extended the scope of existing mutation testing and provided more fault detection [48]. A limitation of this study can be attributed to its exclusive concentration on Java code and the limitation imposed on methods based on a 50-token boundary. Finally, Mastropaolo et al. have recommended a technique that generates mutants using the pre-trained T5 language model. In this approach, they employed a dataset produced by Tufano et al. The experimental results showed that the T5 model generated better mutants than the RNN model used by Tufano et al. [49]. The limitation of this study is the need for measurements regarding the performance of a model trained from scratch for the relevant task, as the study relies solely on a pre-trained T5 model.

When we investigated similar studies in the literature, we observed certain shortcomings. These include the evaluation metrics used for the generated mutant code, low token counts, and the use of pre-trained models. Furthermore, all these studies have been conducted using the Java programming language. In our research, we have identified and addressed these shortcomings. We also introduce the first study conducted with the Python programming language to the literature.

B. NMT IN SOFTWARE ENGINEERING

NMT is an automatic translation system that uses machine learning and deep learning techniques to translate written text from one language to another [50], [51]. This system allows people to facilitate communication instantly. NMT-based models are trained using large amounts of natural language text data. In addition to natural language, there are studies on source code to perform software engineering tasks.

Gao and colleagues proposed a code summarization method that considers the code's structure to help developers understand programs and extract their functionality [52]. Nguyen et al. introduced an NMT based on encoder-decoder architecture to enable the reuse of existing API calls and source code while developing a software project. The presented method has been evaluated on two Android datasets and achieved high accuracy in suggesting API calls and source code [53]. Developers need to replace old versions with new versions to use third-party libraries, and selecting the correct version can be a complex process. Nguyen et al. presented an NMT-based method to learn library upgrades performed by similar clients. This method predicts future versions of relevant libraries and advises the developer on which version of the library to upgrade [54]. Phan et al. recommended a tool considering the API Call Programming Abbreviation context. This tool uses natural language descriptions in the source code to obtain accurate and compilable API calls. The study finds the surrounding code and provides information about local variable names in the code suggestion, reducing programming time [55]. Soliman et al. offered a technique for generating Python source code from natural language explanations [56]. Liguori et al. developed an approach for automatically generating “shellcode,” which exploits software vulnerabilities to assist in investigating and preventing security analysts against malicious attacks. This method automatically produces shellcodes in assembly language starting from natural language descriptions [57]. Reading and understanding bytecode is more difficult for humans compared to source code. Huang et al. proposed a method for generating code annotations to help programmers understand the meaning of bytecode more quickly and accurately during software development activities. This method has demonstrated the feasibility of developing natural language annotations from bytecode [58]. Stein and Mancoridis utilized semantic clues in Python source code to generate meaningful explanations

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

from source code snippets [59]. Writing proper log messages is an essential task for developers. Ding et al. introduced an approach to automatically create brief log messages from source code. In this approach, the source code is taken as input, and the log message is produced considering other contextual information, such as the position of the log statement [60]. A method for automatically explaining errors in code to developers has been presented by Mahbub et al. This approach defines the mistake using the source code's structural information and faulty patterns. According to Google code standards, the study showed that the method produced understandable and reasonable explanations [61]. Rai et al. offered an NMT method that translates Python source code into pseudocode [62]. Pseudo-code written in natural language helps junior developers in the process of understanding programs. However, writing this type of pseudo-code takes time and effort. Therefore, Yang et al. proposed a technique that converts source code to pseudo-code [63]. Li et al. introduced the idea of combining a search-based automatic program repair technique with an NMT-based approach [64]. Ye et al. presented an NMT-based program repair method that includes program-specific information during weight optimization using a loss function based on program compilation and test execution information [65]. Jiang et al. recommended a technique for automatically correcting buggy codes by generating developer-like source code. In this method, they designed a new code-aware search strategy focusing on correctable fixes with a length close to the buggy code and compilable fixes, which find more accurate fixes [66]. Various data-driven and well-trained model-based approaches have emerged to automate software engineering tasks. With improved models and training methods, the performance of these tasks has continuously improved. However, creating a high-quality dataset is both time-consuming and challenging. To address data inadequacy issues, Ahmed and Devanbu provided evidence that software code written in different languages performing the same function is similar, particularly in preserved identifier naming patterns [67]. Roziere et al. proposed a method for unsupervised code translation using automated unit tests. The authors leveraged the presence of unit tests in code repositories to create a parallel dataset for training a machine translation model. They used a combination of data augmentation techniques and self-training to improve the quality of the translation. The experimental results demonstrated that their approach achieves promising performance, outperforming existing unsupervised code translation methods on several benchmark datasets [68]. Bui et al. presented a unified framework for learning to debug code using CodeT5, a state-of-the-art model for code generation tasks. The authors introduced the Detect-Localize-Repair (DLR) framework, which consists of three stages: bug detection, localization, and repair. They leveraged the powerful text-to-text transfer learning capabilities of CodeT5 to perform these tasks. Experimental results on two benchmark datasets demonstrated that the proposed framework achieves significant improvements in bug detection, localization, and repair compared to existing baselines. The DLR framework shows promise in automating the debugging process and facilitating more efficient software development [69]. Lutellier et al. introduced a novel approach called Encore for automatic program repair using ensemble learning and convolutional neural machine translation. The authors proposed using a combination of multiple translation models trained on different representations of the input program to improve repair accuracy. They also introduced a novel repair synthesis technique that generates diverse and plausible patches for the detected bugs. Experimental evaluations on benchmark datasets showed that Encore outperforms existing state-of-the-art program repair techniques in terms of repair accuracy and diversity. The results suggested that ensemble learning and convolutional neural machine translation can effectively enhance automatic program repair [70]. Xia et al. discussed the advancements and implications of large pre-trained language models in the field of automated program repair. The authors explored how the emergence of these models, such as GPT-3 and CodeT5, has revolutionized program repair by leveraging their capabilities in natural language understanding and code generation. They provided insights into the challenges and opportunities of applying pre-trained language models to program repair tasks [71]. Lutellier et al. presented Coconut, an approach that combines context-aware neural translation models through ensemble learning for program repair. The authors proposed a hierarchical ensemble framework that integrates multiple translation models at different levels of granularity to improve repair accuracy. Coconut incorporated context information from the surrounding code and leveraged attention mechanisms to capture relevant program dependencies. Experimental evaluations on benchmark datasets demonstrated that Coconut outperforms existing program repair techniques in terms of both repair accuracy and diversity of generated patches. The results highlighted the effectiveness of ensemble learning and context-aware translation models for program repair tasks [72]. Hata et al. introduced a method for learning to generate corrective patches using NMT techniques. The authors proposed a framework that treats program repair as a translation task, where buggy code was translated into its fixed version. They trained an NMT model using a parallel dataset consisting of buggy and fixed code examples. Experimental results on benchmark datasets showed that their approach achieves competitive performance in generating accurate and meaningful patches for program repair. The study highlighted the potential of leveraging NMT for automated program repair tasks [73].

III. APPROACH

We start by mining bug fixes commits from thousands of GitHub repositories (Section III-A1). We obtain source code

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

files from these bug fixes commits in Python (Section III-A2). We extract method-level pairs of the buggy and corresponding fixed code (Section III-A3). We categorize these method-level code pairs according to the type of mutation (Section III-A4). Then, using an Abstract Syntax Tree, we abstract these categorized method-level code pairs into a representation more suitable for learning (Section III-A5). All special tokens were carefully preserved through the use of a mapping technique during the abstraction process. Finally, for the abstracted method-level code pairs, we employ a Transformer-based encoder-decoder model to enable the transformation of a fixed code snippet into its corresponding buggy version (Section III-B). The entire flow diagram of the system is shown in Fig. 1.

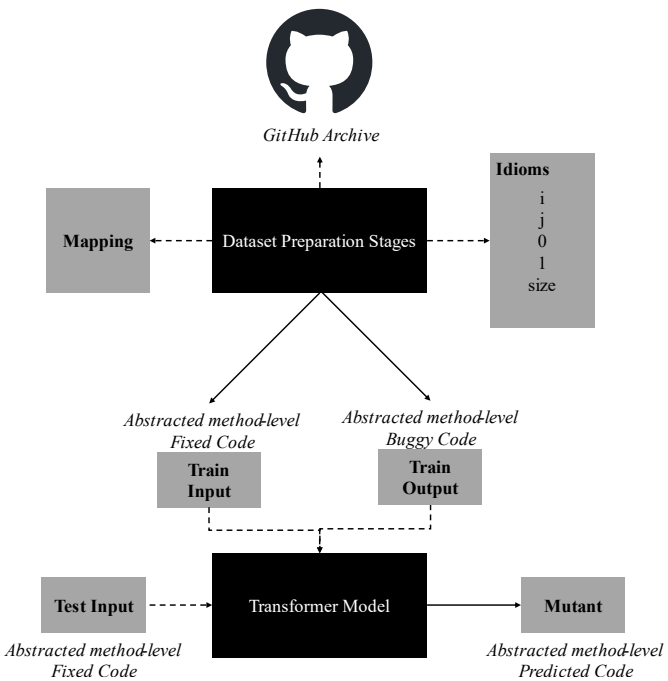


FIGURE 1. The pipeline of the workflow and architecture of the proposed approach

A. DATASET

This section outlines the steps in creating the dataset. Initially, we mine bug fixes commits from a multitude of GitHub repositories. From these commits, we procure source code files written in Python. We then isolate pairs of buggy code and the corresponding fixed code at the method-level. These pairs are subsequently classified based on the nature of the mutation. Finally, by employing an Abstract Syntax Tree, we transform these categorized method-level code pairs into a more conducive learning form. During the abstraction process, a mapping technique was employed to ensure the preservation of all special tokens. The entire relationship between these stages is shown in Fig. 2.

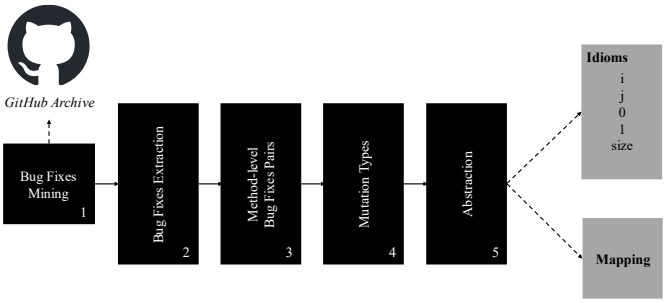


FIGURE 2. Overview of dataset creation approach for the mutant generation

1) BUG FIXES MINING FROM GITHUB

First, we downloaded the GitHub Archive [74], which contains public Python programming language events from January 2011 to October 2021. Next, we used Google BigQuery APIs [75] to identify commits related to bug fixes. Finally, we selected commits containing patterns like ("fix" or "solve") and ("bug" or "issue" or "problem" or "error") in all messages [76]. As a result, we identified 5,864,922 commits associated with bug fixes, from which we extracted the commit ID (SHA) and project repository name.

Not all commit matching these message patterns are related to bug fixes in the Python programming language. Changes in other programming languages within the project repository may also match these patterns. Additionally, the commit may differ from those genuinely related to bug fixes. Therefore, two authors collaborated to determine the accuracy and precision of the commits. Upon examining a total of 500 randomly selected samples, we observed that 27 of them were not related to bug fixes. We analyzed the statistical confidence level of the commits at 94.6%, with a ±5% confidence interval. Due to the high confidence level, we used all commits in our study.

2) BUG FIXES EXTRACTION

Using GitHub's Compare API, we extracted the before and after (buggy and fixed) source code of commits related to bug fixes. (i) Multiple commits can be made to a source code file in the project repository. Therefore, we used the latest commit to the relevant source code file as the buggy and fixed one and ignored the others. (ii) Since our goal is to learn the mutation operators of the Python language, we only used the commits made to the Python source code in the project repository. As a result of these filtering processes, we extracted 588,016 buggy and fixed source code files from commits associated with bug fixes. Fig. 3 shows the buggy and fixed source code obtained using the GitHub Compare API. In the commit differences, lines indicated with '+' are added to the fixed source code file, while lines marked with '-' are added to the buggy source code file. Fig. 4 a) and b) illustrates separated buggy and fixed source code files.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

```
    def factorial( number ) :
        # Calculates the factorial of a given number
        if number == 0 :
            return 1
        else:
-           fact = 0
+           fact = 1
-           for num in range ( 1 , number ) :
+           for num in range ( 1 , number + 1 ) :
                fact = fact * num
            return fact

    if __name__ == " __main__ " :
        num = 3
        print ( " The factorial of " , num , " is " , factorial ( num ) )
```

FIGURE 3. Committed source code obtained using GitHub Compare API

```
    def factorial( number ) :
        # Calculates the factorial of a given number
        if number == 0 :
            return 1
        else:
-           fact = 0
-           for num in range ( 1 , number ) :
                fact = fact * num
            return fact

    if __name__ == " __main__ " :
        num = 3
        print ( " The factorial of " , num , " is " , factorial ( num ) )
```

a) Buggy source code

```
    def factorial( number ) :
        # Calculates the factorial of a given number
        if number == 0 :
            return 1
        else:
+           fact = 1
+           for num in range ( 1 , number + 1 ) :
                fact = fact * num
            return fact

    if __name__ == " __main__ " :
        num = 3
        print ( " The factorial of " , num , " is " , factorial ( num ) )
```

b) Fixed source code

FIGURE 4. Extraction of bug fixes pairs

3) METHOD-LEVEL BUG FIXES PAIRS

We created method-level pairs from both buggy and fixed source code files. These pairs will be used to train a Transformer-based model to generate mutants that resemble actual bugs. However, the source code files contained snippets extending beyond the method-level. Therefore, we first extracted methods from the buggy code files and their corresponding fixed code files. Each file can contain multiple methods, but the corresponding file might not include a method code snippet. For this reason, we filtered out such cases and extracted the corresponding buggy and fixed code pairs at the method-level. As a result, we obtained 2,277,408 method-level buggy and fixed code pairs. Fig. 5 illustrates the method-level code pairs obtained from the source code files.

```
    def factorial( number ) :
        # Calculates the factorial of a given number
        if number == 0 :
            return 1
        else:
            fact = 0
            for num in range ( 1 , number ) :
                fact = fact * num
            return fact
```

a) Method-level buggy code

```
    def factorial( number ) :
        # Calculates the factorial of a given number
        if number == 0 :
            return 1
        else:
            fact = 1
            for num in range ( 1 , number + 1 ) :
                fact = fact * num
            return fact
```

b) Method-level fixed code

FIGURE 5. Method-level bug fixes pairs

4) MUTATION TYPES

We identified three mutation types for generating mutants similar to actual bugs from the method-level buggy and fixed code pairs: Update, Delete, and Insert. We developed a syntax-aware diff tool to classify the mutation types in the method-level buggy and fixed code pairs. This tool detects deleted, added, moved, and renamed code nodes between the method-level buggy code and the corresponding fixed code. A node refers to a discrete part of the code, such as a statement, expression, function, or block. Nodes are used to analyze and manipulate code during the mutation process. We represented the differences in method-level code pairs with four nodes: DeleteNode for node deletion operations, InsertNode for node additions, MoveNode for node moving, and UpdateAttrib for node modifications. We classified method-level code pairs by mutation types based on the count of these nodes in the code. We generated the differences between method-level code pairs based on the fixed code by corresponding with the nodes in the buggy code—the Update mutation type forms when all nodes indicating differences in the code pair are UpdateAttrib. For the Delete mutation type to form, all nodes must be DeleteNode. In the Insert mutation type, all nodes are InsertNode. Method-level code pairs may contain multiple node types. To ensure accurate classification, we eliminated all code pairs containing more than one node type. As a result of these filtering processes, we obtained 371,378 method-

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

level buggy and fixed code pairs for the Update mutation type, 454,084 for the Delete mutation type, and 243,580 for the Insert mutation type. Node actions occurring in method-level code pairs are illustrated in Fig. 6.

```
[
    UpdateAttrib ( node = ' … ' , name = ' label ' , value = ' 0 ' ) ,
    MoveNode ( node = ' … ' , target = ' … ' , position = 2 ) ,
    DeleteNode( node = ' … ' ) ,
    DeleteNode( node = ' … ' ) ,
    DeleteNode( node = ' … ' )
]
```

FIGURE 6. Mutant nodes

5) ABSTRACTION

Due to the potentially unlimited number of identifiers and literals, machine learning-based models generally need help with large vocabulary sizes in source code. To reduce vocabulary size, an abstraction is applied. Various studies exist that focus on the implementation of the abstraction process. Jiang et al. proposed a deep learning-based automatic code repair approach to Java source code. First, they abstracted identifiers and literals in the source code, including data types, function names, variable names, and constant values, to unique tokens. Subsequently, three new special tokens were added to the vocabulary for unknown identifiers [77]. Hu et al. introduced a deep learning-based method for measuring code maintainability. This method replaced meaningful tokens in the source code with meaningless tokens to reduce the number of identifiers [78]. Zhou and colleagues presented a meta-learning-based approach for code summarization. They performed abstraction using a special token <num> to replace numerical constants in the source code [79]. Wang et al. recommended a transformer-based binary code similarity detection technique. In this technique, during abstraction, string constants were replaced with <str>, constant variables with <const>, and internal functions with <function> [80]. Zhang et al. conducted various software engineering tasks using CoditT5, a pre-trained language model on source code and natural language [81]. They utilized the src2abs tool developed by Tufano and colleagues [82] for the Java programming language to perform automatic code reviews and reduce vocabulary size. The abstraction tool replaces identifiers, literals, and function names with special tokens [83]. Izadi et al. offered a deep learning-based automatic code completion approach. First, they removed comments and blank lines during preprocessing to improve code quality. Then, they abstracted identifiers, literals, and function names in the Python dataset with special tokens [84].

The issue with raw method-level buggy and fixed code pairs is that they can have unlimited identifiers and constant literals, making the vocabulary size incredibly large. Large-scale vocabulary size is problematic for generating mutants like actual bugs using NMT. Therefore, we performed an abstraction process to reduce the vocabulary size. We decided to replace raw tokens in the code with unique tokens. We used the Python-specific Abstract Syntax Tree (AST) [85], [86] to find and replace these unique tokens. We applied the abstraction process first to the fixed code and then to the buggy code. Finally, the method-level fixed code is input into NMT to generate mutants similar to actual bugs. The expected output is the method-level buggy code. Abstracted method-level buggy and fixed code pairs are illustrated in Fig. 7 a) and b).

```
def METHOD_1 ( VAR_1 ) :
    if VAR_1 == 0 :
        return 1
    else :
        VAR_2 = 0
        for VAR_3 in range ( 1 , VAR_1 ) :
            VAR_2 = VAR_2 * VAR_3
        return VAR_2
```

a) Abstracted method-level buggy code with idioms

```
def METHOD_1 ( VAR_1 ) :
    if VAR_1 == 0 :
        return 1
    else :
        VAR_2 = 1
        for VAR_3 in range ( 1 , VAR_1 + 1 ) :
            VAR_2 = VAR_2 * VAR_3
        return VAR_2
```

b) Abstracted method-level fixed code with idioms

FIGURE 7. Abstracted method-level bug fixes pairs

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

First, we removed docstrings and comments from the code pairs. Then, we assigned special tokens to identifiers and constant literals in a consecutive and position-dependent manner. The first method in the syntax receives the METHOD_1 special token, and the second method receives using a mapping method. The abstracted codes were generated from NMT, and with the mapping, it became possible to revert to the raw token values. Lastly, we limited our dataset's token count of the abstracted method-level buggy and fixed code pairs to 100.

```
def METHOD_1 ( VAR_1 ) : NEWLINE INDENT if VAR_1 == 0 : NEWLINE INDENT return 1
NEWLINE DEDENT else: NEWLINE INDENT VAR_2 = 0 NEWLINE for VAR_3 in range ( 1 , VAR_1 )
: NEWLINE INDENT VAR_2 = VAR_2 * VAR_3 NEWLINE DEDENT return VAR_2
```

a) Abstracted method-level buggy inline code with idioms

```
def METHOD_1 ( VAR_1 ) : NEWLINE INDENT if VAR_1 == 0 : NEWLINE INDENT return 1
NEWLINE DEDENT else: NEWLINE INDENT VAR_2 = 1 NEWLINE for VAR_3 in range ( 1 , VAR_1
+ 1 ) : NEWLINE INDENT VAR_2 = VAR_2 * VAR_3 NEWLINE DEDENT return VAR_2
```

b) Abstracted method-level fixed inline code with idioms

FIGURE 8. Abstracted inline method-level bug fixes pairs

the METHOD_2 token. This process continues for all method and variable names (VAR_X, ATTRIBUTE_X, EXCEPTION_X, ANNOTATION_X, and KEYWORD_X) and constant literals (STRING_X, INT_X, FLOAT_X, BYTES_X) within the code. Note that special tokens are shared between code pairs and new special tokens are only created for new identifiers or constant literals. The special tokens generated for abstracted code significantly reduced the number of unique words in the vocabulary size, as they can be reused across different code pairs. For example, when performing the abstraction process for a new code pair, the first method found will be replaced with METHOD_1 again. Some identifiers and constant literals frequently appear in source code and can almost be treated as keywords of the language for abstraction purposes. For example, loops often use variables such as i, j, or index. Similarly, conditional statements and return values commonly use constants such as 0, 1, and -1. Method names like “get” are frequently used in code to represent common concepts. These identifiers and constant literals are often referred to as “idioms.” We retain idioms in our representation, meaning we do not substitute idioms with a generated identifier but instead preserve the original text in the code representation. To create the list of idioms, we begin by sampling 1 million method-level pairs and examining all of their source code. We then extract the frequency of each identifier/literal used in the code, discarding keywords, separators, and comments. Next, we analyze the distribution of the frequencies and focus on the most frequently used words. Specifically, we concentrate on the top 500 words in the distribution. Finally, we added NEWLINE, INDENT, and DEDENT tokens to the code pairs to ensure the usability and syntactic correctness of the generated code when replacing tokens with special tokens. After incorporating these special tokens, the code pairs were used as inline code, as illustrated in Fig. 8 a) and b). All special tokens involved in the abstraction process were stored

B. TRANSFORMER MODEL

The Transformer is built upon the encoder-decoder architecture, commonly employed for tasks such as NMT, where a sequence of words is translated from one language to another [87]. This architecture comprises two main components: (i) the encoder, which transforms an input sequence of tokens into a series of embedding vectors, frequently referred to as the hidden state or context, and (ii) the decoder, which, utilizing the hidden state produced by the encoder, iteratively generates an output sequence of tokens, creating one token at each step. First, the input text undergoes tokenization and is transformed into token embeddings. Since the attention mechanism does not inherently recognize the relative positions of tokens, it is necessary to incorporate information about token positions into the input to account for the sequential nature of the text. Therefore, token embeddings are combined with positional embeddings, providing each token's positional information. The encoder comprises a series of encoder layers or “blocks,” while the decoder has its own set of stacked decoder layers. The output from the encoder is supplied to each decoder layer, producing a prediction for the most likely subsequent token in the sequence. The outcome of this process is then input back into the decoder to generate the following token. This continues until a special end-of-sequence (<EOS>) token is encountered. Attention is a mechanism that enables neural networks to allocate varying weight or “attention” levels to each element within a sequence. The core concept is that, rather than relying on a fixed embedding for each token, the entire sequence can be utilized to calculate a weighted average of each embedding. There are four primary steps to implement this mechanism: (i) transform each token embedding into three distinct vectors called query, key, and value, (ii) calculate attention scores, (iii) determine attention weights; and (iv) update the token

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

embeddings. First, the whole model, including the encoder and decoder, is trained end-to-end by minimizing the negative log-likelihood of the target terms, utilizing stochastic gradient descent. Fig. 9 shows the architecture of the Transformer model mentioned.

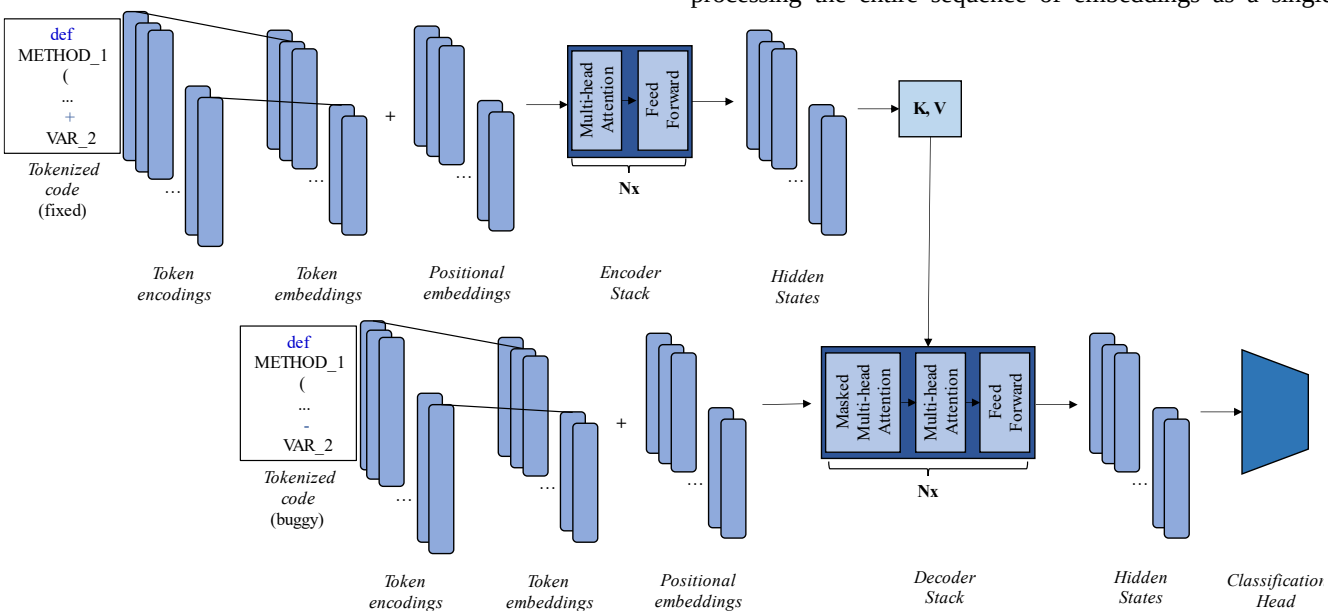


FIGURE 9. Transformer model architecture

IV. EXPERIMENTAL SETTINGS

We investigate how bug fixes can contribute to learning and generating mutants. We develop models using three distinct datasets and one common dataset: Update, Delete, Insert, and General. The General model is trained with each dataset's abstracted method-level buggy and fixed code pairs. Dataset distributions are provided in Table 1. The datasets are divided into 70% for training, 20% for validation, and 10% for testing.

TABLE 1. Dataset distribution

Model	Train (70%)	Validation (20%)	Test (10%)
General	128,847	36,813	18,409
Update	45,312	12,946	6,474
Delete	59,999	17,143	8,572
Insert	23,536	6,724	3,363

We utilized an encoder-decoder-based transformer model to generate buggy code from fixed code at the method-level. The bug fixes pairs are tokenized and converted to 512-token embeddings. The token embeddings are merged with sinusoidal positional embeddings to provide positional information for each token. The encoder of the transformer consists of six layers arranged sequentially, while the decoder comprises six stacked decoder layers. Multi-head attention allows the model to focus on information from various representation subspaces and distinct positions simultaneously. Our approach utilizes eight parallel attention layers, also known as heads. The feed-forward sublayer in the encoder and decoder is a two-layer, fully connected neural network with a unique characteristic. Instead of processing the entire sequence of embeddings as a single vector, it processes each embedding individually. As a result, this layer is commonly known as a position-wise feed-forward layer. The input and output dimensions are 512, and the inner layer has a dimensionality of 2048. We employed the Adam optimizer with $\beta_1 = 0.9$, $\beta 2 = 0.98$, and $\varepsilon = 1e\text{-}9$. We used the Weights & Biases [88] tool to explore the model's hyperparameters during training. The exploration process was conducted using a random search. We determined that the appropriate hyperparameters were a batch size of 128 and a learning rate of 3e-4 for finding the best model. We trained our models using one machine with 8 NVIDIA Quadro P5000 GPUs. We used the hyperparameters mentioned in the paper. Each mutation type's model was trained for over 36 hours, while the general model was trained for 2.5 days.

V. EXPERIMENTAL DESIGN AND RESULTS

This study aims to generate neural machine NMT-based mutants by learning from bug fixes made in a dataset of bug fixes mined from Python open-source projects hosted on GitHub (see Section III). The study aims to measure the quality of the mutant code by evaluating it against the five criteria described below.

1) BLEU SCORE

We employ the Bilingual Evaluation Understudy (BLEU) score as our initial performance measurement, a widely-used metric for evaluating the quality of machine-generated

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

translations [89]. The reference text is needed to determine the similarity between the candidate and reference texts to calculate the BLEU score. Both texts are divided into n-grams, and the algorithm calculates the number of n-grams from the candidate text that appear in the reference text. Finally, we present the BLEU score, representing the geometric mean of all n-grams up to a length of four. We first calculate the BLEU score between the method-level abstracted fixed code and its corresponding target buggy code to evaluate our mutant generation method. This BLEU score acts as a baseline for comparison. We then compute the BLEU score between the predicted mutant and the target buggy code. Finally, the BLEU score is measured between the abstracted fixed code at the method level and the predicted mutation code. A higher BLEU score indicates a more remarkable similarity between the abstract predicted code and the abstract buggy code, representing the actual buggy code. To fully comprehend the degree of similarity between our prediction and the existing buggy code, we must compare the BLEU score to our established baseline. The input code (i.e., the fixed code) given to our model could be regarded as a "close translation" of the buggy code, thereby contributing to a high BLEU score. To prevent bias, we compare the BLEU score between the fixed and buggy code (baseline) to the BLEU score obtained when comparing the predicted mutant with the actual buggy code. Suppose the BLEU score between predicted and buggy is higher than between fixed and buggy. In that case, it implies that the model has transformed the input code (fixed) into a mutant (predicted) closer to the buggy code than before the mutation. In other words, the mutation moves in the correct direction. Conversely, if the predicted code is further from the buggy code than the original input, it represents a translation that has moved in the wrong direction. Additionally, if the calculated BLEU score between the predicted mutant code and the fixed code is lower than these scores, it is said that a valid mutant code has been generated. To determine if the differences in BLEU scores between the baseline and the models are statistically significant, we utilize a confidence interval method similar to Zhang et al. [90]. We generate m = 2,000 test sets by sampling with replacement from the given test set. We then evaluate the BLEU score on the m-test sets for our model and the baseline. Next, we calculate the m deltas of the scores: model − baseline. Based on the distribution of the deltas, we select the 95% confidence interval (from the 2.5th percentile to the 97.5th percentile). The differences are considered statistically significant if the confidence interval is above or below zero (e.g., 2.5th percentile < 0).

TABLE 2. Comparison of BLEU scores

Model	$abstract_f$ - $abstract_b$ (baseline)	$abstract_p$ - $abstract_b$ (mutated)	$abstract_f$ - $abstract_p$ (validator)
General	89.23	89.11	90.21
Update	96.31	96.23	94.62
Delete	80.29	90.24	84.63
Insert	83.13	85.43	83.53

Table 2 presents the BLEU scores of the general model and other models: the baseline, the mutated, and the validator. The results indicate that the models outperform the baseline when the source code associated with the buggy target code is mutated. Furthermore, the significant difference in BLEU scores compared to the baseline suggests that the mutants are generated in accordance with the expected bugs. Based on the reported results, the BLEU score indicates that our proposed approach generates code predictions that are more similar to the buggy code.

TABLE 3. Comparison of BLEU scores with confidence interval

Model	2.5^{th} percentile mutated - baseline
General	+0.55
Update	+0.11
Delete	+3.14
Insert	+2.0

Table 3 presents the BLEU scores of the 2.5th percentile of the delta distribution. The results demonstrate that the models outperform the baseline when the source code related to the buggy target code is mutated. The score differences are statistically significant, and the 2.5th percentile of the delta distribution (+0.55) indicates that the models' BLEU scores are slightly higher than those of the baseline. This improvement in the BLEU score suggests that our approach predicts code closer to the buggy code than the fixed code.

2) PREDICTION CLASSIFICATION

Given the abstracted fixed ($abstract_f$), abstracted buggy ($abstract_b$), and abstracted predicted code ($abstract_p$), we categorize each prediction into one of the following groups: (i) perfect prediction, when abstracted predicted equals abstracted buggy (meaning the model has successfully reverted the fixed code to its buggy version, reintroducing the original bug); (ii) bad prediction, when abstracted predicted equals abstracted fixed (indicating that the model failed to mutate the code and returned the same input code); and (iii) mutated prediction, when abstracted predicted is not equal to the abstracted buggy and abstracted predicted is not equal to abstracted fixed (signifying that the model mutated the code differently than the target buggy code). Finally, we present the raw numbers and percentages of predictions that fall into these specified categories.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

TABLE 4. Comparison of predicted classes

Model	Perfect pred.	Mutated pred.	Bad pred.	Total
General	2,373 (12.9%)	12,816 (69.6%)	3,220 (17.5%)	18,409
Update	1,193 (18.4%)	2,309 (35.7%)	2,972 (45.9%)	6,474
Delete	3,014 (35.2%)	4,879 (56.9%)	679 (7.9%)	8,572
Insert	198 (5.9%)	2,948 (87.7%)	217 (6.4%)	3,363

The resemblance of the generated mutants to the actual buggy code is shown in Table 4. The generated mutant code resembles existing buggy code ranging from 5% to 35%. General, the system produces mutated code at a rate of 80%. However, when each dataset is examined separately, it is observed that the system performs better than the general dataset.

3) SYNTACTIC CORRECTNESS

To ensure the effectiveness of mutants, they must be syntactically correct, allowing the project to be compiled and tested against the test suite. We use a Python AST grammar to assess the lexical and syntactical correctness of the model's predictions. Perfect and incorrect predictions are already known to be syntactically correct, as we verified the correctness of the buggy and fixed code when extracting the method-level buggy and fixed code pairs. However, the correctness of predictions in the mutated prediction category remains uncertain. As a result, we provide the percentages of syntactically correct predictions and those for mutated predictions. It is important to note that we do not evaluate the compilability of the code.

TABLE 5. Comparison of syntactic correctness

Model	Mutated pred.	Overall
General	97.46%	98.03%
Update	98.70%	99.18%
Delete	97.93%	98.46%
Insert	95.66%	96.19%

Table 5 demonstrates that the model accurately predicts over 96% of sentences in terms of both lexical and syntactic accuracy. Furthermore, even when considering mutated predictions, the model maintains a high level of syntactic correctness (>95%). These results suggest the model learns the appropriate syntax rules from the abstracted representation used as input/output.

4) TOKEN-BASED OPERATIONS

We also analyzed the models' predictions based on the operations performed on their tokens and categorized them into four types: (i) insertion, where the number of tokens in the prediction is more significant than that in the input; (ii) changes, where the number of tokens in the prediction is the same as that in the input, but the prediction differs from the input; (iii) deletions, where the number of tokens in the prediction is fewer than that in the input; and (iv) none, where the prediction is the same as the input. This analysis helps evaluate the models' ability to insert, modify, or delete tokens.

TABLE 6. Comparison of token-based operations

Model	Insertion	Changes	Deletion	None
General	1,759 (9.6%)	7,343 (39.7%)	6,087 (33.2%)	3,220 (17.5%)
Update	2 (0.1%)	3,493 (53.9%)	7 (0.1%)	2,972 (45.9%)
Delete	17 (0.2%)	128 (1.5%)	7,748 (90.4%)	679 (7.9%)
Insert	2,749 (81.7%)	324 (9.6%)	73 (2.3%)	217 (6.4%)

We analyzed the token operations performed during mutant generation and created three different datasets for each type. Each dataset performs token operations with a specific purpose, as shown in Table 6. The Delete and Insert mutation models perform token operations for their intended purpose 81% of the time. However, the Update mutation model achieves its goal only 53% of the time due to producing similar code to the fixed code. The general model performs token operations according to the distribution rate in the dataset since it includes each type of mutant.

5) AST-BASED OPERATIONS

We prioritize evaluating mutated predictions to determine if they represent an improvement. However, our primary goal is to ascertain whether the model's transformations resemble those required to convert the fixed code into the corresponding buggy code. Specifically, we investigate whether the model applies AST actions similar to the fixed and buggy code transformations. As a result, we report both the percentages of mutated predictions that share the same elements and frequency of actions as those in the fixed-to-buggy transformation, as well as the percentages of mutated predictions that contain the same unique actions, albeit possibly in a different order, location, or frequency than those in the perfect prediction (buggy code).

TABLE 7. Comparison of AST-based operations

Model	Same Operation Set	Same Operation List
General	36.23%	18.82%
Update	98.03%	96.80%
Delete	73.44%	66.83%
Insert	64.50%	56.05%

During the conversion process from fixed to buggy source code, various AST-based operations are performed, resulting

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

in changes to the code snippets. Table 7 shows the quality of these syntax operations for each mutation type. For example, the Update mutation type produces pairs of fixed and buggy code that are syntactically similar, with a 98% similarity rate for the AST-based operation performed. However, the complexity of the operations increases as the mutation types become more complicated, such as Delete and Insert. As a result, the general model, which includes each mutation type, predicts the corresponding buggy mutant at a low syntactic rate.

6) MUTANT GENERATION

We used the Greedy Search algorithm to convert the abstracted fixed ($abstract_f$) given as input into mutant code. Greedy Search, also known as Greedy Decoding [91], is a simple decoding method used in various natural language processing tasks such as machine translation, and text generation. The basic idea of Greedy Search is to make the most likely choice at each step in the generation process. More formally, given a sequence of tokens or words, Greedy Search selects the token with the highest conditional probability given the previous tokens at each step, and then it moves onto the next token. We have carried out the previous five evaluations mentioned under the main heading, according to the codes generated from the decoding strategy under this heading.

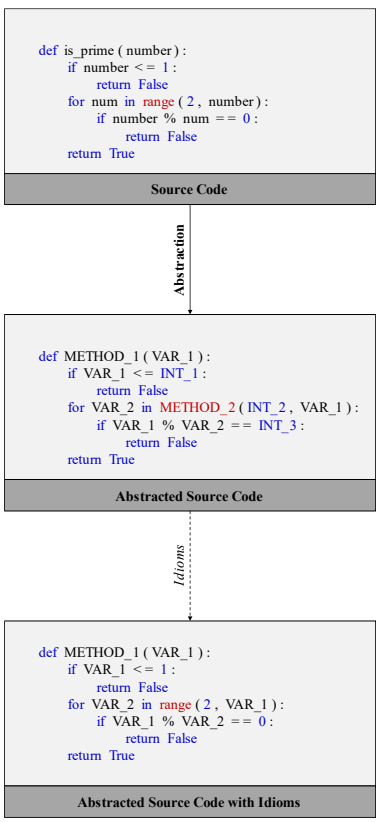


FIGURE 10. The stages of converting the source code into input code for the Transformer model

The process of converting method-level source code into mutant code involves several steps. Firstly, abstraction is applied to the source code snippet. During this process, identifiers and constant literals in the code are replaced with special identifiers. These special identifiers in the abstracted code are then substituted with the most frequently occurring idioms in the dataset. After these steps, the input code for the Transformer model is obtained. Fig. 10 illustrates the application of these stages for identifying prime numbers at the method level.

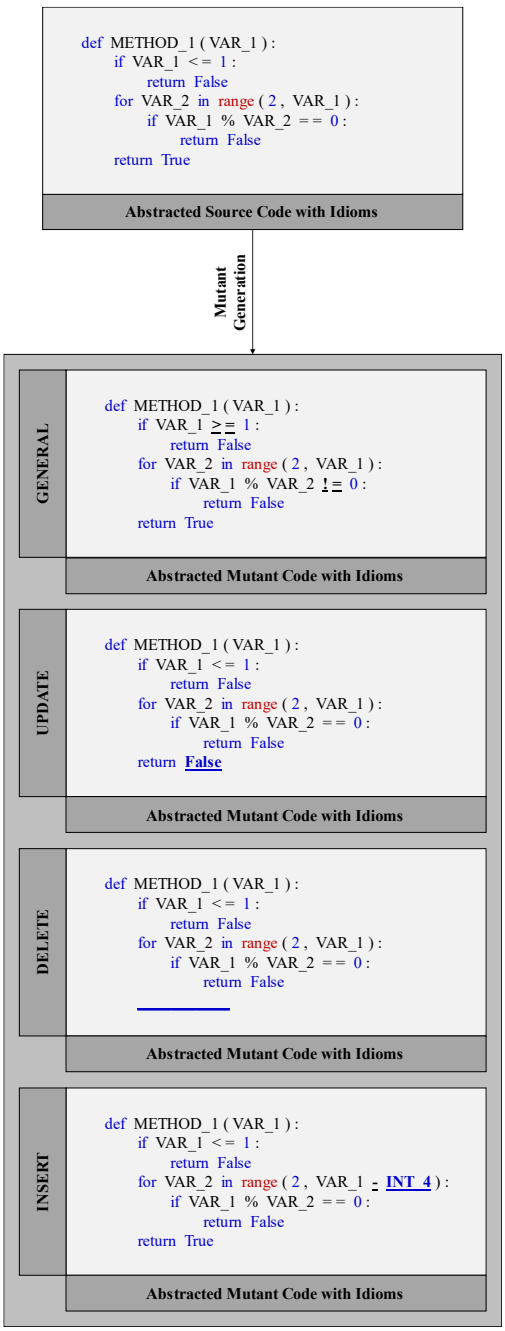


FIGURE 11. The examples of generated qualitative mutants

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

After the preprocessing stages, the source code is primed for input in the Transformer-based NMT model. This model transforms the input source code into mutant code. Fig. 11 presents qualitative mutant examples generated from the preprocessed code snippet for each model type. The general model introduces changes to two comparison operators. The first operator change prompts the prime number-finding method to return incorrect results for all integers greater than 1. In the second operator change, if the remainder of the division operation differs from zero, the prime number-finding method returns an incorrect result. The Update model modifies just one line of code, which causes the prime number finding function to return incorrect results consistently. The Delete model has wholly removed a line from the code, causing the prime number finding method not to return any result when the prime number condition is satisfied. Consequently, the condition must be fulfilled for integers meeting the prime number criteria. Finally, the Insert model has added constant literals within the code. These constant literals alter the range of the loop in the prime number-finding method, causing the loop to deviate from the expected range and leading to the incorrect operation of the code.

VI. THREATS TO VALIDITY

Concerning *external validity*, this study aims to generate mutants solely for bug fixes related to the Python programming language. The absence of a similar study conducted with this programming language demonstrates that we are undertaking one of the most comprehensive works in the field of mutant code generation for Python. Projects with varying sizes and different management structures can yield different results. However, it cannot be guaranteed that these findings can be generalized to different programming languages.

With respect to *construct validity*, the bug fixes dataset we collected from the past may contain mistakes. For example, a commit can be reverted, but we do not take such intentions into account. Additionally, we used message patterns to identify bug fix commits (Section III-A3). We obtained the bug fixes dataset based on these message patterns. However, the bug fixes dataset can be used not only for mutant code generation but also for program repair.

Another threat to *construct validity* exists in our preprocessing. We classified the source code based on mutation types. This stage was developed through trial-and-error experiments and evaluated experimentally. However, different methods and processes can enhance the classification based on mutation types. Further exploration of the preprocessing configuration should be beneficial for future studies.

To mitigate threats to *reliability* and enhance transparency, we are openly providing our dataset [92] and code [93] to everyone. By making the dataset accessible to a wider audience, our aim is to foster collaboration, encourage replication studies, and facilitate the verification and validation of our findings. This open approach ensures that researchers and practitioners can independently examine the data, assess its quality, and draw their own conclusions, thereby strengthening the reliability of our work. Additionally, by openly sharing the dataset, we enable the research community to build upon our findings, explore novel perspectives, and contribute to the advancement of knowledge in the field.

VII. DISCUSSION

The data obtained from our experimental results confirm that we have achieved the primary goal of our study: the feasibility of producing mutants that resemble real-world bugs. Each model generates mutants successfully, according to the type of mutation. Furthermore, during the generation of mutants, the syntactic correctness of the code snippet is ensured with high accuracy. Despite these successes, our study has several areas for improvement. Firstly, the mutant codes we obtained are abstracted code snippets. This could result in the generation of abstracted identifiers that differ from those in the input code snippets. Consequently, it might not always be feasible to revert these snippets to their original identifier values. Secondly, the mutant code snippets we have generated are non-executable. Lastly, due to the model's limitation of a 100-token count, we're prevented from using larger code snippets (those with a token count exceeding 100). These challenges could be addressed by increasing hardware resources, avoiding the abstraction of identifiers within the code snippets, and expanding the model's capacity.

VIII. CONCLUSION

We presented an NMT-based approach for automatically learning mutants from existing bug fixes. The aim of this method is to generate mutants that closely resemble actual bugs present in the source code. The generated mutants exhibit a range of fidelity, with success rates ranging from 6% to 35% (depending on the dataset and model) in successfully reintroducing the original bugs into the fixed code. This means that our models are able to effectively revert the fixed code back to its buggy version, essentially reproducing the original bug. This is a significant achievement because it demonstrates the capability of our models to understand the underlying logic and characteristics of bugs and accurately reproduce them. Furthermore, our models have the ability to generate different variations of buggy code, deviating from the original bug. In fact, they are able to produce the actual bug and other types of buggy code up to 94% of the time. This showcases the versatility of our approach in generating a wide range of mutants, providing researchers with a valuable resource for studying and analyzing bugs in the Python programming language. An interesting aspect of our approach is that our models are able to learn correct code syntax without relying on explicit

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

syntax rules as input. This implies that the models developed an implicit understanding of the syntax through the training process, enabling them to generate mutants that adhere to the expected syntax conventions of Python. To facilitate further research in this area, we have made the generated data available to other researchers. By sharing this dataset, we aim to encourage collaboration and enable the use of our mutants in various projects related to bug detection, program analysis, and software testing [92]. While our work has yielded promising results in automatically generating mutants for Python, it is crucial to emphasize that the development of a comprehensive infrastructure is essential for effectively injecting these mutants. Injecting mutants into code requires careful consideration of factors such as test coverage, test case generation, and integration with existing software development workflows. Therefore, the practical implementation of our approach requires the establishment of a complete infrastructure that can seamlessly incorporate the mutants produced by our models. In future work, our aim is to enhance the ability to recognize and repair a wider range of bugs in Automatic Program Repair by effectively utilizing machine learning and deep learning techniques. We will utilize the dataset we have created, which is related to bug fixes. Additionally, as part of our future research, we plan to create a dataset specifically for testing deep learning models. The testing process for the deep learning model will be conducted similarly to mutation testing, involving variations in model layers, activation functions, and other factors. Moreover, our ultimate goal is to transform these two future studies into an online platform that can be utilized by users.

ACKNOWLEDGMENT

This work was supported by the VALU3S project, which has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No. 876852. The JU receives support from the European Union's Horizon 2020 research and innovation program and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, and Turkey (TUBITAK, under contract no:120N800). The views expressed in this work are the authors' sole responsibility and do not necessarily reflect those of the European Commission. The authors, the VALU3S Consortium, and the ECSEL JU are not responsible for any use that may be made of the information contained herein.

REFERENCES

[1] M. K. Thota, F. H. Shajin, and P. Rajesh, "Survey on software defect prediction techniques," *International Journal of Applied Science and Engineering*, vol. 17, no. 4, pp. 331–344, Jan. 2020, doi: 10.6703/IJASE.202012_17(4).331.

[2] O. Taiwo and A. E. Ezugwu, "Internet of Things-Based Intelligent Smart Home Control System," *Security and Communication Networks*, vol. 2021, 2021, doi: 10.1155/2021/9928254.

[3] N. Y. R. Douha, M. Bhuyan, S. Kashihara, D. Fall, Y. Taenaka, and Y. Kadobayashi, "A survey on blockchain, SDN and NFV for the smart-home security," *Internet of Things*, vol. 20, p. 100588, Nov. 2022, doi: 10.1016/J.IOT.2022.100588.

[4] G. Alexakis, S. Panagiotakis, A. Fragkakis, E. Markakis, and K. Vassilakis, "Control of Smart Home Operations Using Natural Language Processing, Voice Recognition and IoT Technologies in a Multi-Tier Architecture," *Designs 2019, Vol. 3, Page 32*, vol. 3, no. 3, p. 32, Jul. 2019, doi: 10.3390/DESIGNS3030032.

[5] E. Guclu, Ö. Örnek, M. Ozkan, A. Yazici, and Z. Demirci, "An Online Distance Tracker for Verification of Robotic Systems' Safety," *Sensors*, vol. 23, no. 6, p. 2986, 2023.

[6] M. Ozkan, Z. Demirci, Ö. Aslan, and A. Yazıcı, "Safety Verification of Multiple Industrial Robot Manipulators with Path Conflicts Using Model Checking," *Machines 2023, Vol. 11, Page 282*, vol. 11, no. 2, p. 282, Feb. 2023, doi: 10.3390/MACHINES11020282.

[7] H. V Gamido and M. V Gamido, "Comparative review of the features of automated software testing tools," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, no. 5, pp. 4473–4478, 2019, doi: 10.11591/ijece.v9i5.pp4473-4478.

[8] B. H. Thacker, S. W. Doebling, F. M. Hemez, M. C. Anderson, J. E. Pepin, and E. A. Rodriguez, "Concepts of Model Verification and Validation." 2004. Accessed: May 10, 2023. [Online]. Available: http://inis.iaea.org/Search/search.aspx?orig_q=RN:36030870

[9] D. R. Wallace and R. U. Fujii, "Software Verification and Validation: An Overview," *IEEE Softw*, vol. 6, no. 3, pp. 10–17, 1989, doi: 10.1109/52.28119.

[10] J. M. Balera and V. A. de Santiago Júnior, "A systematic mapping addressing Hyper-Heuristics within Search-based Software Testing," *Inf Softw Technol*, vol. 114, pp. 176–189, Oct. 2019, doi: 10.1016/J.INFSOF.2019.06.012.

[11] G. Petrović and M. Ivanković, "State of mutation testing at google," *Proceedings - International Conference on Software Engineering*, pp. 163–171, May 2018, doi: 10.1145/3183519.3183521.

[12] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just, "Practical Mutation Testing at Scale: A view from Google," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3900–3912, Oct. 2022, doi: 10.1109/TSE.2021.3107634.

[13] M. Beller *et al.*, "What it would take to use mutation testing in industry - A study at facebook," *Proceedings - International Conference on Software Engineering*, pp. 268–277, May 2021, doi: 10.1109/ICSE-SEIP52600.2021.00036.

[14] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just, "Does mutation testing improve testing practices?," *Proceedings - International Conference on Software Engineering*, pp. 910–921, May 2021, doi: 10.1109/ICSE43902.2021.00087.

[15] A. Panichella and C. C. S. Liem, "What Are We Really Testing in Mutation Testing for Machine Learning? A Critical Reflection," *Proceedings - International Conference on Software Engineering*, pp. 66–70, May 2021, doi: 10.1109/ICSE-NIER52604.2021.00022.

[16] S. J. Kaufman, R. Featherman, J. Alvin, B. Kurtz, P. Ammann, and R. Just, "Prioritizing Mutants to Guide Mutation Testing," *Proceedings - International Conference on Software Engineering*, vol. 2022-May, no. 22, pp. 1743–1754, 2022, doi: 10.1145/3510003.3510187.

[17] B. Arasteh, P. Imanzadeh, K. Arasteh, F. S. Gharehchopogh, and B. Zarei, "A Source-code Aware Method for Software Mutation Testing Using Artificial Bee Colony Algorithm," *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 38, no. 3, pp. 289–302, Jun. 2022, doi: 10.1007/S10836-022-06008-9/TABLES/9.

[18] M. B. Kusharki, S. Misra, B. Muhammad-Bello, I. A. Salihu, and B. Suri, "Automatic Classification of Equivalent Mutants in Mutation Testing of Android Applications," *Symmetry 2022, Vol. 14, Page 820*, vol. 14, no. 4, p. 820, Apr. 2022, doi: 10.3390/SYM14040820.

[19] A. Derezińska and K. Hałas, "Analysis of mutation operators for the Python language," *Advances in Intelligent Systems and*

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

Computing, vol. 286, pp. 155–164, 2014, doi: 10.1007/978-3-319-07013-1_15/COVER.

[20] Electronic Components and Systems, "Strategic Research and Innovation Agenda," *Erişim:*, 2023. https://www.smart-systems-integration.org/publication/ecs-sria-2023 (accessed May 07, 2023).

[21] D. Bahdanau, K. H. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, Sep. 2014, Accessed: May 07, 2023. [Online]. Available: https://arxiv.org/abs/1409.0473v7

[22] R. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer (Long Beach Calif)*, vol. 11, no. 4, pp. 34–41, 1978, doi: 10.1109/C-M.1978.218136.

[23] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," *Advances in Computers*, vol. 112, pp. 275–378, Jan. 2019, doi: 10.1016/BS.ADCOM.2018.03.015.

[24] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," *ISSTA 2017 - Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, vol. 12, pp. 295–306, Jul. 2017, doi: 10.1145/3092703.3092714.

[25] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption," *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pp. 597–608, Jul. 2017, doi: 10.1109/ICSE.2017.61.

[26] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 354–365, Jul. 2016, doi: 10.1145/2931037.2931040.

[27] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," *Proceedings - International Conference on Software Engineering*, vol. 14-22-May-2016, pp. 523–534, May 2016, doi: 10.1145/2884781.2884791.

[28] F. Hariri, A. Shi, H. Converse, S. Khurshid, and D. Marinov, "Evaluating the Effects of Compiler Optimizations on Mutation Testing at the Compiler IR Level," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pp. 105–115, Dec. 2016, doi: 10.1109/ISSRE.2016.51.

[29] M. Kintis, M. Papadakis, and N. Malevris, "Employing second-order mutation for isolating first-order equivalent mutants," *Software Testing, Verification and Reliability*, vol. 25, no. 5–7, pp. 508–535, Aug. 2015, doi: 10.1002/STVR.1529.

[30] M. Kusano and C. Wang, "CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications," *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pp. 722–725, 2013, doi: 10.1109/ASE.2013.6693142.

[31] H. Yoshida, S. Tokumoto, M. R. Prasad, I. Ghosh, and T. Uehara, "FSX: Fine-grained incremental unit test generation for C/C++ programs," *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 106–117, Jul. 2016, doi: 10.1145/2931037.2931055.

[32] J. Nanavati, F. Wu, M. Harman, Y. Jia, and J. Krinke, "Mutation testing of memory-related operators," *2015 IEEE 8th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2015 - Proceedings*, May 2015, doi: 10.1109/ICSTW.2015.7107449.

[33] F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke, "Memory mutation testing," *Inf Softw Technol*, vol. 81, pp. 97–111, Jan. 2017, doi: 10.1016/J.INFSOF.2016.03.002.

[34] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. Garc\'\ia-Dom\'\inguez, and J. J. Dom\'\inguez-Jiménez, "Corrigendum to 'Assessment of class mutation operators for C++ with the MuCPP mutation system'[Information and Software Technology, 81,(2017) 169–184]," *Inf Softw Technol*, vol. 100, no. 87, p. 103, 2017.

[35] A. Parsai, A. Murgia, and S. Demeyer, "LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10522 LNCS, pp. 148–163, 2017, doi: 10.1007/978-3-319-68972-2_10/TABLES/5.

[36] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using conditional mutation to increase the efficiency of mutation analysis," *Proceedings - International Conference on Software Engineering*, pp. 50–56, 2011, doi: 10.1145/1982595.1982606.

[37] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for Java," *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, pp. 433–436, Jul. 2014, doi: 10.1145/2610384.2628053.

[38] P. R. Mateo, M. P. Usaola, and J. Offutt, "Mutation at system and functional levels," *ICSTW 2010 - 3rd International Conference on Software Testing, Verification, and Validation Workshops*, pp. 110–119, 2010, doi: 10.1109/ICSTW.2010.18.

[39] L. Madeyski and N. Radyk, "Judy - A mutation testing tool for Java," *IET Software*, vol. 4, no. 1, pp. 32–42, 2010, doi: 10.1049/IET-SEN.2008.0038/CITE/REFWORKS.

[40] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for Java (Demo)," *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 449–452, Jul. 2016, doi: 10.1145/2931037.2948707.

[41] E. Omar, S. Ghosh, and D. Whitley, "HOMAJ: A tool for higher order mutation testing in AspectJ and Java," *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2014*, pp. 165–170, 2014, doi: 10.1109/ICSTW.2014.19.

[42] P. Madiraju and A. S. Namin, "Paraμ- A partial and higher-order mutation tool with concurrency operators," *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, pp. 351–356, 2011, doi: 10.1109/ICSTW.2011.34.

[43] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," *2013 International Symposium on Software Testing and Analysis, ISSTA 2013 - Proceedings*, pp. 224–234, 2013, doi: 10.1145/2483760.2483773.

[44] M. Gligoric, V. Jagannath, and D. Marinov, "MuTMuT: Efficient exploration for mutation testing of multithreaded code," *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation*, pp. 55–64, 2010, doi: 10.1109/ICST.2010.33.

[45] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "ProFIPy: Programmable Software Fault Injection as-a-Service," *Proceedings - 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020*, pp. 364–372, Jun. 2020, doi: 10.1109/DSN48063.2020.00052.

[46] "ESOGU-SRLAB/imfit," *Erişim:*, 2023. https://github.com/ESOGU-SRLAB/imfit (accessed May 07, 2023).

[47] R. Degiovanni and M. Papadakis, "μBert: Mutation Testing using Pre-Trained Language Models," *Proceedings - 2022 IEEE 14th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2022*, pp. 160–169, 2022, doi: 10.1109/ICSTW55395.2022.00039.

[48] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Learning How to Mutate Source Code from Bug-Fixes," *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pp. 301–312, Sep. 2019, doi: 10.1109/ICSME.2019.00046.

[49] A. Mastropaolo *et al.*, "Using Transfer Learning for Code-Related Tasks," *IEEE Transactions on Software Engineering*, 2022, doi: 10.1109/TSE.2022.3183297.

[50] W. Zhang, L. Dai, J. Liu, and S. Wang, "Improving Many-to-Many Neural Machine Translation via Selective and Aligned

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

Online Data Augmentation," *Applied Sciences 2023, Vol. 13, Page 3946*, vol. 13, no. 6, p. 3946, Mar. 2023, doi: 10.3390/APP13063946.

[51] H. Chen, Y. Chen, and J. Zhang, "Neural Machine Translation of Electrical Engineering Based on Vector Fusion," *Applied Sciences 2023, Vol. 13, Page 2325*, vol. 13, no. 4, p. 2325, Feb. 2023, doi: 10.3390/APP13042325.

[52] S. Gao *et al.*, "Code Structure–Guided Transformer for Source Code Summarization," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–32, Feb. 2023, doi: 10.1145/3522674.

[53] P. T. Nguyen, C. Di Sipio, J. Di Rocco, D. Di Ruscio, and M. Di Penta, "Fitting missing API puzzles with machine translation techniques," *Expert Syst Appl*, vol. 216, p. 119477, Apr. 2023, doi: 10.1016/J.ESWA.2022.119477.

[54] P. T. Nguyen, J. Di Rocco, R. Rubei, C. Di Sipio, and D. Di Ruscio, "DeepLib: Machine translation techniques to recommend upgrades for third-party libraries," *Expert Syst Appl*, vol. 202, p. 117267, Sep. 2022, doi: 10.1016/J.ESWA.2022.117267.

[55] H. Phan, A. Sharma, and A. Jannesari, "Generating Context-Aware API Calls from Natural Language Description Using Neural Embeddings and Machine Translation," *Proceedings - 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASEW 2021*, pp. 219–225, 2021, doi: 10.1109/ASEW52652.2021.00050.

[56] A. S. Soliman, M. M. Hadhoud, and S. I. Shaheen, "MarianCG: a code generation transformer model inspired by machine translation," *Journal of Engineering and Applied Science*, vol. 69, no. 1, pp. 1–23, Dec. 2022, doi: 10.1186/S44147-022-00159-4/TABLES/9.

[57] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Can we generate shellcodes via natural language? An empirical study," *Automated Software Engineering*, vol. 29, no. 1, pp. 1–34, May 2022, doi: 10.1007/S10515-022-00331-3/TABLES/9.

[58] Y. Huang, J. Huang, X. Chen, K. He, and X. Zhou, "BCGen: a comment generation method for bytecode," *Automated Software Engineering*, vol. 30, no. 1, pp. 1–31, Jun. 2023, doi: 10.1007/S10515-022-00374-6/FIGURES/9.

[59] A. Stein and S. Mancoridis, "Chaos to Clarity with Semantic Inferencing for Python Source Code Snippets," in *IEEE International Conference on Semantic Computing (ICSC'23)*, 2023.

[60] Z. Ding, H. Li, and W. Shang, "LoGenText: Automatically Generating Logging Texts Using Neural Machine Translation," *Proceedings - 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022*, pp. 349–360, 2022, doi: 10.1109/SANER53432.2022.00051.

[61] P. Mahbub, O. Shuvo, and M. M. Rahman, "Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation," *arXiv preprint arXiv:2212.04584*, 2022.

[62] S. Rai, R. C. Belwal, and A. Gupta, "Is the Corpus Ready for Machine Translation? A Case Study with Python to Pseudo-Code Corpus," *Arab J Sci Eng*, vol. 48, no. 2, pp. 1845–1858, Feb. 2023, doi: 10.1007/S13369-022-07049-0/TABLES/3.

[63] G. Yang, Y. Zhou, X. Chen, and C. Yu, "Fine-grained Pseudo-code Generation Method via Code Feature Extraction and Transformer," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2021-December, pp. 213–222, 2021, doi: 10.1109/APSEC53868.2021.00029.

[64] D. Li, W. E. Wong, M. Jian, Y. Geng, and M. Chau, "Improving Search-Based Automatic Program Repair With Neural Machine Translation," *IEEE Access*, vol. 10, pp. 51167–51175, 2022, doi: 10.1109/ACCESS.2022.3164780.

[65] H. Ye, M. Martinez, and M. Monperrus, "Neural Program Repair with Execution-based Backpropagation," *Proceedings - International Conference on Software Engineering*, vol. 2022-May, pp. 1506–1518, 2022, doi: 10.1145/3510003.3510222.

[66] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," *Proceedings - International Conference on Software Engineering*, pp. 1161–1173, May 2021, doi: 10.1109/ICSE43902.2021.00107.

[67] T. Ahmed and P. Devanbu, "Multilingual training for Software Engineering," *Proceedings - International Conference on Software Engineering*, vol. 2022-May, pp. 1443–1455, 2022, doi: 10.1145/3510003.3510049.

[68] B. Rozière, J. M. Zhang, F. Charton, M. H. Facebook, G. Synnaeve, and G. Lample, "Leveraging Automated Unit Tests for Unsupervised Code Translation," Oct. 2021, Accessed: Jul. 07, 2023. [Online]. Available: https://arxiv.org/abs/2110.06773v2

[69] N. D. Q. Bui, Y. Wang, and S. C. H. Hoi, "Detect-Localize-Repair: A Unified Framework for Learning to Debug with CodeT5," *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 812–823, Nov. 2022, Accessed: Jul. 07, 2023. [Online]. Available: https://arxiv.org/abs/2211.14875v3

[70] T. Lutellier, L. Pang, V. H. Pham, M. Wei, and L. Tan, "ENCORE: Ensemble Learning using Convolution Neural Machine Translation for Automatic Program Repair," Jun. 2019, Accessed: Jul. 07, 2023. [Online]. Available: https://arxiv.org/abs/1906.08691v1

[71] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models".

[72] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining context-aware neural translation models using ensemble for program repair," *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 101–114, Jul. 2020, doi: 10.1145/3395363.3397369.

[73] H. Hata, E. Shihab, and G. Neubig, "Learning to Generate Corrective Patches using Neural Machine Translation," Dec. 2018, Accessed: Jul. 07, 2023. [Online]. Available: https://arxiv.org/abs/1812.07170v2

[74] "GH Archive," *Erişim:*, 2023. https://www.gharchive.org/ (accessed May 07, 2023).

[75] "BigQuery Enterprise Data Warehouse," *Erişim:*, 2023. https://cloud.google.com/bigquery (accessed May 07, 2023).

[76] S. Zafar, M. Z. Malik, and G. S. Walia, "Towards Standardizing and Improving Classification of Bug-Fix Commits," *International Symposium on Empirical Software Engineering and Measurement*, vol. 2019-Septemer, Sep. 2019, doi: 10.1109/ESEM.2019.8870174.

[77] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "Knod: Domain knowledge distilled tree decoder for automated program repair," *arXiv preprint arXiv:2302.01857*, 2023.

[78] Y. Hu, H. Jiang, and Z. Hu, "Measuring code maintainability with deep neural networks," *Front Comput Sci*, vol. 17, no. 6, pp. 1–15, Dec. 2023, doi: 10.1007/S11704-022-2313-0/METRICS.

[79] Z. Zhou, H. Yu, G. Fan, Z. Huang, and K. Yang, "Towards Retrieval-Based Neural Code Summarization: A Meta-Learning Approach," *IEEE Transactions on Software Engineering*, pp. 1–24, Jan. 2023, doi: 10.1109/TSE.2023.3238161.

[80] H. Wang *et al.*, "JTrans: Jump-aware transformer for binary code similarity detection," *ISSTA 2022 - Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, vol. 22, pp. 1–13, Jul. 2022, doi: 10.1145/3533767.3534367.

[81] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "CoditT5: Pretraining for Source Code and Natural Language Editing," *ACM International Conference Proceeding Series*, vol. 12, no. 22, Sep. 2022, doi: 10.1145/3551349.3556955.

[82] M. Tufano, M. Di Penta, C. Watson, M. White, G. Bavota, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 832–837, Sep. 2018, doi: 10.1145/3238147.3240732.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3302695

[83] M. Tufano, "micheletufano/src2abs," *Erişim:*, 2019. https://github.com/micheletufano/src2abs (accessed May 07, 2023).
[84] M. Izadi, R. Gismondi, and G. Gousios, "CodeFill: Multi-token Code Completion by Jointly learning from Structure and Naming Sequences," *Proceedings - International Conference on Software Engineering*, vol. 2022-May, pp. 401–412, 2022, doi: 10.1145/3510003.3510172.
[85] "Abstract Syntax Trees — Python documentation." https://docs.python.org/3.8/library/ast.html (accessed May 07, 2023).
[86] "Python AST Explorer." https://python-ast-explorer.com/ (accessed May 07, 2023).
[87] A. Vaswani *et al*., "Attention is All you Need," *Adv Neural Inf Process Syst*, vol. 30, 2017.
[88] "Weights & Biases." https://wandb.ai/site (accessed May 07, 2023).
[89] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.
[90] Y. Zhang, S. Vogel, and A. Waibel, "Interpreting bleu/nist scores: How much improvement do we need to have a better system?," in *LREC*, 2004.
[91] U. Germann, "Greedy decoding for statistical machine translation in almost linear time," in *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 2003, pp. 72–79.
[92] "ChArIoT-Dataset," *Erişim:*, 2023. https://drive.google.com/drive/folders/1n_IRc_8zT_1n3HL1sj_IXEQlcgDWfWIj (accessed May 07, 2023).
[93] Aşık Sergen and U. Yayan, "ESOGU-SRLAB/ChArIoT-Testing." https://github.com/ESOGU-SRLAB/ChArIoT-Testing (accessed Jul. 12, 2023).



SERGEN AŞIK received the B.S. degree in computer engineering from the Eskisehir Osmangazi University, Turkey, where he is currently pursuing the M.S. degree in computer engineering. His main research interests include natural language processing and software reliability.



UĞUR YAYAN After graduating as an Electrical and Electronics Engineer (EEE) in 2009, he received his Master's degree in EEE from Eskisehir Osmangazi University (ESOGU) in 2012. He obtained his Ph.D. in Control and Command program in EEE from ESOGU in 2018. After completing his Ph.D., he started working as an Assistant Professor in the Department of Software Engineering at ESOGU. He is also the Vice-Director of the ESOGU Intelligence Factory and Robotics Laboratory Digital Innovation Hub (IFARLAB-DIH) and the Director of ESOGU Autonomous Systems and Reliability Laboratory (ASRLAB).

Since 2010, he has participated in 11 national and 9 international funded projects. Additionally, he has 3 patents and applications, 3 design registrations, and 3 utility models. Moreover, he has authored 18 international articles, 17 international conference proceedings, and 30 national conference proceedings. He has also written 2 books and 2 book chapters on robotics.

His areas of interest include reliability estimation, manufacturing, robotics, route planning, digital twins, artificial intelligence, verification and validation, and mutation testing.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4