Carl von Ossietzky
Universität
Oldenburg

Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Neural Bug Detection

Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels eines

Doktor der Naturwissenschaften (Dr. rer. nat.)

angenommene Dissertation von

Cedric Richter
geboren am 04.01.1996 in Duisburg

Gutachter
Prof. Dr. Heike Wehrheim,
Prof. Dr. Lars Grunske
Tag der Disputation:
16.05.2025

Abstract

Software bugs, especially small mistakes that manifest in code, cost developers and companies a lot of time and money. If they make it into production code, software bugs can have a serious impact on our daily software-driven lives. Therefore, to mitigate the risks of bugs in code, software developers are often interested in finding and fixing software bugs as early as possible in the development process.

Neural bug detectors have been recently proposed as a way to automatically support software developers in their debugging efforts. The idea is to learn from millions of examples consisting of buggy and bug-free programs to detect novel bugs in code. Neural bug detectors are already effective in detecting simple bugs such as wrong binary operators or variables that are mistakenly used. However, existing neural bug detectors have still many limitations that make them difficult to employ in the practice. For example, recent studies have found that neural bug detectors miss a significant amount of real bugs while producing a high number of false alarms. Therefore, currently, software developers have to sift through many false alarms before they find an alarm corresponding to a real bug.

The goal of this thesis is to address the current limitations of existing neural bug detectors. We hypothesize that existing neural bug detectors are limited by two factors: (1) the training process and (2) the task design. To be able to train neural bug detectors on millions of examples, existing work often relies on artificially generated mutants for training. The mutants are generated by performing random replacements. We hypothesize that the generated mutants are not representative for real bugs which could explain the low performance of neural bug detectors on real bugs. To evaluate this hypothesis, we employ two strategies. First, we propose a novel contextual mutation operator that can generate more realistic mutants. Second, we mine public repositories for real bug fixes that represent real bugs found in open source projects. We show that training both on more realistic mutants and on real bug fixes can significantly improve the performance of neural bug detectors on real bugs. Our second hypothesis is that neural bug detectors are limited by their task design. Existing neural bug detectors are often designed for function-level bug detection. We hypothesize that because of the missing context neural bug detector make mistakes that would have been otherwise preventable. To collect first evidence for this hypothesis, we developed and evaluated an LLM-based validator that can make use of extra context typically not available to a neural bug detector. Based on our research results, we end up with a neural bug detector design that is significantly more effective in detecting real bugs and in avoiding false alarms.

Zusammenfassung

Software Bugs, vor allem kleine Fehler in Programmen, kosten Entwickler und Unternehmen viel Zeit und Geld. Wenn sie es in den Code der Computersysteme schaffen, die wir täglich benutzen, dann können Softwarefehler auch ernsthafte Auswirkungen auf unser tägliches Leben haben. Daher sind Softwareentwickler häufig daran interessiert, Softwarefehler so früh wie möglich im Entwicklungsprozess zu finden und zu beheben.

Neural Bug Detectors wurden vor kurzem vorgeschlagen, um Softwareentwickler die Fehlersuche zu erleichtern. Die Idee ist, aus Millionen von Beispielen fehlerhafter und fehlerfreier Programme zu lernen, wie man neue Softwarefehler in Programmen erkennt. Existierende Neural Bug Detectors sind bereits jetzt schon sehr effektiv bei der Erkennung von einfachen Fehlern, wie zum Beispiel bei der Erkennung von fälschlich verwendeten binären Operatoren oder Variablen. Allerdings haben existierende Neural Bug Detectors noch viele Einschränkungen, die Ihren Einsatz in der Praxis erschweren. Zum Beispiel zeigen jüngere Studien, dass Neural Bug Detectors noch viele echte Fehler übersehen und gleichzeitig ein hohe Anzahl von Fehlalarmen produzieren.

In dieser Dissertation stellen wir die Hypothese auf, dass Neural Bug Detectors (1) durch das Design des Trainingsprozesses und (2) durch die Definition der Aufgabenstellung beschränkt sind. Um genügend Trainingsdaten für das Training von Neural Bug Detectors zu haben, benutzen aktuelle Arbeiten noch künstlich erzeugte Fehler. Die Fehler werden dadurch erzeugt, dass der Programmcode zufällig mutiert wird. In den meisten Fällen sind die daraus resultierenden Fehler nicht repräsentativ für die tatsächlichen echten Fehler, die in der Praxis beobachtet werden. Da die Neural Bug Detectors häufig nur die Fehler findet, die sie auch im Training sehen, könnte der Einsatz von künstlichen Fehlern im Training dazu führen, dass echte Fehler nicht erkannt werden. Um diese Hypothese zu überprüfen, wenden wir zwei Strategien an. Unsere erste Strategie ist die Entwicklung eines neuen kontextbezogenen Mutationsoperators, welcher in der Lage ist, realistischere Mutationen in Programmen zu erzeugen. Unsere zweite Strategie ist das Durchsuchen von öffentlichen Repositories nach echten "Bug Fixes", welche echte Fehler (Bugs) und deren Fixes darstellen. Wir zeigen, dass das Training mit realistischeren Mutanten und echten Fehlern die Effektivität von Neural Bug Detectors bei der Erkennung von echten Fehlern deutlich erhöht. Unsere zweite Hypothese ist, dass Neural Bug Detectors durch ihr eigenes Aufgabendesign begrenzt sind. Bestehende Neural Bug Detectors werden oft für das Erkennen von Fehlern in Funktionen entwickelt. Es könnte aber sein, dass Neural Bug Detectors aufgrund des fehlenden Kontextes Fehler machen, die ansonsten vermeidbar gewesen wären. Um erste Hinweise zum Beweisen der Hypothese zu sammeln, haben wir einen LLM-basierten Validator entwickelt und evaluiert, der zusätzlichen Kontext nutzen kann, der einem Neural Bug Detectors normalerweise nicht zur Verfügung steht. Basierend auf unseren Forschungsergebnissen dieser Dissertation war es uns möglich, ein neues Design für Neural Bug Detectors zu entwickeln, das wesentlich effektiver bei der Erkennung echter Fehler und der Vermeidung von Fehlalarmen ist.

Acknowledgment

As the journey of my PhD comes to an end, I would like to express my deepest gratitude and appreciation to all those that have supported and guided me along the way.

First of all, I am profoundly thankful for my supervisor Heike Wehrheim. I am extremely grateful for all your support, advice and for allowing me to pursue my unconventional research interests. You always made time for me, always listened with patience to the many ideas I shared over the years, and offered valuable feedback and new insights when needed. You helped me grow as a researcher, scientist and person.
I also want to thank the members of my thesis committee, Andreas Peter and Friederike Bruns, for their valuable insights. Their feedback and insightful comments have greatly enriched the quality of my thesis. I am deeply grateful for my second reviewer Lars Grunske. His support and genuine interest in my work over the years has helped me stay motivated and his valuable feedback has helped me shape my thesis. Lars is always someone who I am looking forward to see at software engineering conferences.

I would also like to thank my co-authors, collaborators and student assistants which worked with me on so many fascinating projects. I am particularly grateful to Jan Haltermann, Marie-Christine Jakobs, Felix Pauck, Stefan Schott, Dirk Beyer, Sudeep Kanav, Florian Dyck, Christian Janßen, and Marek Chalupa.

I am also grateful to my colleagues Arnab, Jan, Felix, Jürgen, Lara, Manuel and Nicola. Thank you for the great discussions and also for the many opportunities to look at computer science research from a completely different perspective. You made my time in Oldenburg and Paderborn much more enjoyable. In addition, you provided excellent feedback to the initial draft version of this thesis, which helped me a lot to improve its quality. A special thanks to Sven Walther, who introduced me to the research group in 2016. Without you, I might never have discovered the joy of academic research.

Finally, I would like to thank my family, my parents and my two brothers Robin and John, for supporting me throughout my whole life. You encourage and motivate me everyday to pursue my dreams and helped me a lot through difficult times. Thank you for supporting me throughout this experience.

Contents

1 Introduction **1**

1.1 From Static to Neural Bug Detection 2
1.2 Central Hypotheses of this Thesis 3
1.3 Contribution and Outline 6
1.4 Publication Details 7

2 Background **9**

2.1 Software Bugs 9
2.1.1 Real and Artificial Software Bugs 10
2.1.2 Categorizing Software Bugs into Patterns 11
2.2 Neural Models of Code 12
2.2.1 Deep Neural Networks 12
2.2.2 Important Applications of Code Models 16
2.3 Neural Bug Detection 18
2.3.1 Neural Bug Detection and Repair 19
2.3.2 Neural Architecture for Bug Detection and Repair 20
2.3.3 Generating Training Data by Mutating Programs 22

3 Mutations for Neural Bug Detection **25**

3.1 Motivation 25
3.2 Mutation Operators 27
3.3 Contextual Mutations 28
3.3.1 Contextual Mutation Operator 29
3.3.2 Generating Mutation Candidates with a Mask Mutator 30
3.3.3 Contextual Mutant Selection with Language Models 31
3.3.4 Implementation 33
3.4 Evaluation 34
3.4.1 Evaluation Tasks 34
3.4.2 Mutation Operator Types 36
3.4.3 Mutation Operator Baselines 37

3.5 Results . . . 38
3.5.1 RQ1 - Are contextual mutants more realistic? . . . 38
3.5.2 RQ2 - Impact on the training of Neural Bug Detectors . . . 39
3.5.3 RQ3 - Transfer to other languages and bug types . . . 42
3.6 Threats to Validity . . . 44
3.7 Related Work . . . 45
3.8 Conclusion . . . 47

4 Mining Realistic Bugs **49**
4.1 Motivation . . . 49
4.2 Single Statement Bug Fixes in the Wild . . . 50
4.3 Mining Real Bug Fixes at Massive Scale . . . 52
4.3.1 Mining Single Statement Changes in Python Projects . . . 53
4.3.2 True Single Statement Bug Fixes . . . 55
4.3.3 Characterizing Bug Fixing Edits . . . 56
4.4 Dataset Analysis . . . 56
4.4.1 RQ1 - Does the distribution of bug fixes change? . . . 57
4.4.2 RQ2 - How different are bugs that do not classify as SStuBs? . . . 58
4.5 Threats to Validity . . . 60
4.6 Related Work . . . 61
4.7 Conclusion . . . 63

5 Learning from Real Bug Fixes **65**
5.1 Motivation . . . 65
5.2 Neural Bug Detection of Single Token Bugs . . . 66
5.3 Studying the Impact of Real Bug Fixes and Mutants at Scale . . . 67
5.3.1 Training with Code Mutants and Real Bug Fixes . . . 68
5.3.2 Scaling Factors in the Training of Neural Bug Detectors . . . 69
5.3.3 Implementation . . . 70
5.4 Evaluation . . . 70
5.4.1 Evaluation Tasks . . . 71
5.4.2 Neural Bug Detector Baseline . . . 72
5.4.3 Datasets . . . 72
5.5 Results . . . 73
5.5.1 RQ1 - Impact of Real Bug Fixes at Scale . . . 73
5.5.2 RQ2 - Impact of Mutants at Scale . . . 75
5.5.3 RQ3 - Comparison with State of the Art . . . 79
5.6 Discussion . . . 82
5.7 Threats to Validity . . . 84
5.8 Related Work . . . 86
5.9 Contributions and Conclusion . . . 88

6 False Alarm Reduction **91**
6.1 Motivation . . . 91
6.2 Validation for Neural Bug Detection . . . 93
6.3 An LLM-based Validator for Neural Bug Detection . . . 94
6.3.1 Patch Validation with Large Language Models . . . 95
6.3.2 Patch Validation as Selective Code Infilling . . . 96
6.3.3 Adjusting the Validator to Different Contexts . . . 97
6.3.4 Implementation . . . 99
6.4 Evaluation . . . 99
6.4.1 Evaluation Tasks . . . 100
6.4.2 Baselines . . . 101
6.5 Results . . . 102
6.5.1 RQ1 - Validating Bug Fixes and False Alarms . . . 102
6.5.2 RQ2 - Impact of Context . . . 105
6.5.3 RQ3 - Impact on Neural Bug Detection . . . 106
6.5.4 RQ4 - Finding Novel Bugs in Public Projects . . . 108
6.6 Discussion . . . 110
6.7 Threats to Validity . . . 113
6.8 Related Work . . . 115
6.9 Contributions and Conclusions . . . 116

7 Conclusion **119**
7.1 Summary . . . 119
7.2 Discussion and Outlook . . . 120

A Appendix **123**
A.1 Simple Stupid Bug Patterns . . . 123
A.2 Alternative Contextual Mutation Operators . . . 126
A.2.1 Contextual Mutation Operators . . . 126
A.2.2 Evaluation . . . 128
A.2.3 Which mutation strategy is more effective? . . . 128
A.3 Scanning Open Source Projects for Real Bugs . . . 130
A.3.1 Scanned Projects . . . 130
A.3.2 Found Bugs and Quality Issues . . . 131

Bibliography **131**

List of Figures **159**

List of Tables **160**

1 Introduction

Software influences many aspects of our daily lives. We use software to communicate with each other, to carry out financial transactions and it even influences the safety of the vehicles we use to commute every day. Due to its widespread use, even simple mistakes in the implementation of software systems can have enormous consequences. A simple example of an implementation mistake that impacted the security of nearly 1 billion Apple devices in 2014 is the `'goto fail;'` bug [vD14]. The bug impacted the implementation of SSL/TLS and it made many devices susceptible to man-in-the-middle attacks, i.e. the attacker can intercept the communication and steal valuable information. The mistake itself is simple and an abbreviated version of it is shown below.

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
```

Here, the second `goto` which is not affected by the if condition completely bypasses the signature check and the surrounding function returns with `err = 0` ("no error"). As a result, all signatures – both good and bad ones – would be accepted. Although this bug seems to be easy to find in hindsight, it took nearly two years to be identified and fixed [Art].

There are many examples where simple bugs that seem obvious upon reading the code have made it into production code. For example, a Twitter outage in 2016 was caused by simple date formatting mistake [Twi] and the more recent CrowdStrike outage that crashed nearly 8.5 million Windows devices was caused by a simple bug in an out-of-bounds check [Cro]. Although these examples represent extreme cases, they also demonstrate that we need tools and techniques that support developers in the debugging process and help them to identify bugs earlier – before they go into production and impact our lives.

In practice, numerous approaches have been proposed to support developers in finding bugs. For example, automated testing [BB84] helps to identify bugs by running

the software against a set of user-defined test cases. Static analyses over-approximate the behavior of programs and then check the behavior against a set of hand-crafted rules [HP04]. Formal verification proves that a given implementation adheres to a user-provided specification [JM09]. To be able to detect bugs, many of these approaches require some form of *correctness* specification as input from the developer. This can be given either in the form of a test suite or by an explicit formal specification. In any case, the developer has to encode the *intended* behavior of their programs in some explicit form. Unfortunately, developers often have difficulties in providing precise and complete program specifications. Therefore, a developer might have problems with encoding the constraint "the signature should always be checked" as a formal specification. Or, the developer might only test parts of the software that do not trigger the bug. As a result, some bugs – such as the `'goto fail;'` bug or the CrowdStrike bug – remain undetected as the program behavior is under-specified.

1.1 From Static to Neural Bug Detection

In 2004, Hovemeyer and Pugh [HP04] noticed that some bugs can be easily detected *without* requiring an explicit correctness specification. For example, let us consider our initial example again. Even without knowing the correct intended behavior of the program, it is highly *unlikely* that a developer would use the same goto twice in succession. Therefore, with a simple analysis that checks for double gotos, we could already warn the developer about potential mistakes in their code. This observation has led to the development of *static bug detectors* [HP04, ASPK12, CDD+15]. Static bug detectors do not directly check the correctness of programs, but they scan the code for common *bug patterns* that developers likely want to avoid. As a result, static bug detectors can be easily integrated in the development process to warn the developer early about potential mistakes in their code.

Static bug detectors have gained a lot of popularity with tools such as SpotBugs [HP04], Google's ErrorProne [ASPK12] and Facebook's Infer [CDD+15]. These tools have been integrated in the workflows of large companies such as Google [SAE+18], Facebook [CDD+15], Spotify [Inf] and Mozilla [Inf]. Despite their successes, static bug detectors have many known limitations. Because the rules implemented to check for bugs are often very general, they miss a large fraction of bugs that do not fit a bug pattern [HP18, KS20]. In addition, they also produce a large quantity of false alarms. Developers hence have to sift through false alarms that indicate incorrect program behavior, but that are not valid [SFZ11]. These limitations are likely caused by the lack of correctness specifications. Without a specification, static bug detectors have to make very general assumptions about the intended behavior of a given program which leads to both false alarms and missed bugs.

Neural Bug Detection. Neural bug detection [ABK17, PS18, HSS+19, VKM+19, KMBS20, AJFB21, RW22b, HBV22] has been recently proposed as data-driven al-

ternative to static bug detection. Pradel and Sen [PS18] noticed that developers often encode the intended behavior of functions and variables in function and variable names. For example, consider the `pow` function below.

```
def pow( array, k = 2 ):
   for i in range(len(array)):
      array[i] *= k
```

Without having an explicit correctness specification, we can infer from the identifier names that the function should *likely* compute the k-th power over the elements of a given array. And, if we look closely, we will find that the implementer likely made a mistake: The implementer mistakenly used the multiplication operator `*` instead of the power operator `**` to compute the k-th power (two operators that are easy to confuse). Pradel and Sen found that neural bug detectors can be trained on examples of buggy and non-buggy code to detect these types of *name*-based bugs. Since then, many neural bug detectors [HSS+19, VKM+19, KMBS20, AJFB21, RW22b, HBV22] have been proposed that exploit identifier names and code comments to detect bugs in code. These approaches exploit the observation that developers tend to encode the intended behavior of code *implicitly* in the program as a way to make programs more readable and maintainable [ABDS18].

Despite this conceptual advantage, existing neural bug detectors share a lot in common with static bug detectors. Recent studies [HSS+19] have found that existing neural bug detectors often miss a large fraction of real bugs. In addition, they often report a high number of false alarms [AJFB21] on bug-free code. Because neural bug detectors can in principle infer the intended program behavior, we expect them to overcome limitations of existing static bug detectors. Therefore, in this thesis, we aim to answer the following high-level research question:

> What limits neural bug detectors to be effective for real-world code?

1.2 Central Hypotheses of this Thesis

In the following, we present the central hypotheses that have guided our research. The hypotheses are based on a recent study [RHJ+22] we conducted with over 100 software developers. The goal of this study was to compare the performance of neural bug detectors with the performance of software developers on the task of variable misuse detection [ABK17]. We start by shortly introducing the study design. We then present our findings and the resulting hypotheses.

Study Design. The goal of our study is to compare the performance of software developers and neural bug detectors under the similar constraints. Neural bug detectors are often trained to detect *simple bugs* in *function* implementations. Therefore, we chose the task of variable misuse detection [ABK17] in Java functions to evaluate both neural bug detector and software developer. Variable misuse bugs are common in practice

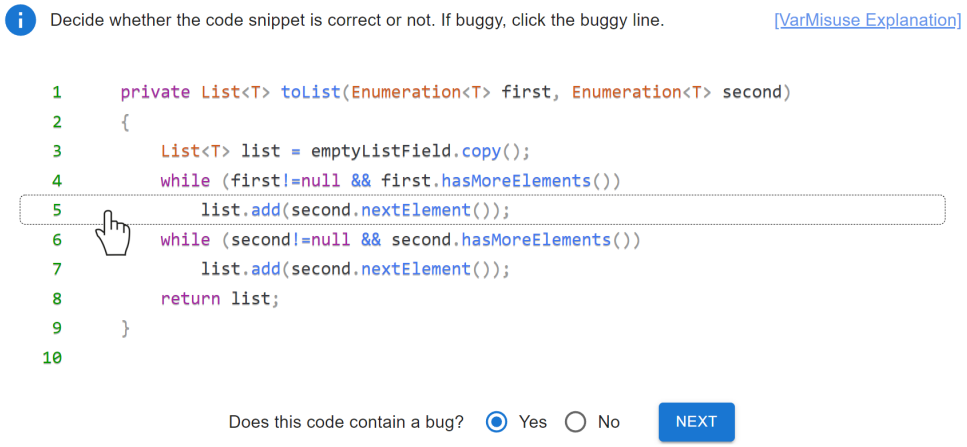


Figure 1.1: User interface of our study with an example of a variable misuse bug

and often occur when the implementer confuses two different variables. An example is shown in Fig. 1.1. The task of variable misuse detection is supported by most neural bug detectors and Allamanis et al. [ABK17] argued that software developers can easily detect this type of bug from experience. To perform our study, we conducted an online survey with over 100 participants. In the survey, software developers were confronted with up to eight Java function implementations and their goal was to decide whether the implementation contains a variable misuse bug and where the bug is located. Out of 310 tasks used in our survey, 134 tasks contained a variable misuse bug which we obtained by mining public Java projects. In total, we collected 1 016 answers from 111 participants for 134 buggy and 176 bug-free functions.

Findings. When the task of variable misuse detection was originally introduced [ABK17], it was believed that software developers can easily detect this type of bug. We however find that:

The task of variable misuse detection is challenging for software developers. We find that the task of finding variable misuse bugs in Java functions is challenging for software developers. On average, the software developers miss more than one third of all variable misuses. Even if they detect a variable misuse, the participants fail on average in more than 50% of all cases to localize the bug. Software developers are better in identifying bug-free functions. However, the participants also report a false alarm in around 10% of all cases.

Under the same constraints software developers perform comparably. We find that under same constraints, i.e. the detection of variable misuses in the implementation of a single function, software developers perform comparably to neural bug detectors. The neural bug detectors often detect and localize a similar number of variable misuses as the participants. However, the neural bug detectors are significantly worse in identifying bug-free functions with a false alarm rate of nearly 32%.

Hypotheses. We believe that our findings have several implications for neural bug

```
1  protected static FrequencySet<String> combineMax(FrequencySet<String> a,
       → FrequencySet<String> b)
     {
2    FrequencySet<String> result = combineAndClip(a, b, 1);
3    for (Map.Entry<String, MutableInt> entry : a.entrySet()) {
4      result.get(entry.getKey()).v = entry.getValue().v;
5    }
6
7   for (Map.Entry<String, MutableInt> entry :  a.entrySet()) {
8      // BUG: a instead of b (Line 7)
9      MutableInt slot = result.get(entry.getKey());
10     slot.v = Math.max(slot.v, entry.getValue().v);
11   }
12
13   return result;
14 }
```

Listing 1.1: Example of copy-and-paste bug missed by neural bug detectors

detection beyond variable misuse detection in Java. In particular, in our study, we investigated the commonalties and differences in the decisions of software developers and neural bug detectors. Based on our investigation, we hypothesize that:

Hypothesis 1: Neural Bug Detectors are limited by their Training Process. Although there is a great overlap of bugs found by software developers and neural bug detectors, there still exists some bugs only the developer could identify. An example is shown in Listing 1.1. As this bug remains undetected by *all* evaluated neural bug detectors, we suspected that this bug type does not appear often in their common training process. In fact, an analysis of the training corpus showed that less than 1% of all training examples represent copy-and-paste bugs, i.e. where the code contains a line that is almost duplicate to the line that contains the bug. Although this specific problem can potentially be resolved by incorporating more examples of this specific bug type, we hypothesize that the occurrence of this training problem hints at a more fundamental problem in the training of neural bug detectors: *There exist some bugs that are frequent in reality but that the neural bug detectors do not see (often) during training.*

Hypothesis 2: Neural Bug Detectors are limited by their Task Design. During our study, we found that there are some bugs that neither the developers nor the neural bug detectors could detect. We hypothesize that these bugs are not detected due to limitations in the task design. There are two potential limitations of the current task design: (1) a natural limitation and (2) a technical limitation. Neural bug detectors and developers are *naturally* limited by the information provided by the code implementer. If the implementer does not provide enough information to detect a bug, both neural bug detector and developer will fail. The second potential limitation is a *technical limitation* which we could likely resolve by adapting the task design. In our study, both neural bug detectors and software developers are restricted to the implementation of a single function in a foreign code base. For software developers, this is however an artificial limitation. In practice, software developers that review code are often familiar with the code base, have access to related code and can review runtime information.

Having access to this type of information would potentially boost the performance of both developers and neural bug detectors.

1.3 Contribution and Outline

With the goal of answering our overarching research question, we evaluate and collect evidence for our two main hypotheses. In this thesis, we mainly address shortcomings of existing neural bug detectors in the training process (Hypothesis 1). In addition, we also identify and address some potential shortcomings of the task design (Hypothesis 2). As a result of our research, we found that we can significantly improve the performance of neural bug detectors by improving their training process and adapting their task design.

As a starting point, we introduce in Chapter 2 the fundamental concepts needed for neural bug detection and we provide a brief overview over state-of-the-art methods for automatic bug detection.

A key problem when training neural bug detectors is to obtain a sufficient number of realistic training examples. Therefore, in Chapter 3, we propose a novel contextual mutation operator which we use to generate training examples for neural bug detectors. The contextual mutation operator can utilize the surrounding context to inject more realistic mutants. We evaluate whether our contextual mutator generates more realistic training examples than traditional mutation operators used in the training of neural bug detectors. Then, based on our findings, we were able to evaluate the impact of more realistic training examples on the performance of neural bug detectors. We found that neural bug detectors trained on more realistic examples are more effective in finding real bugs (which is the first evidence for Hypothesis 1).

Motivated by this finding, we mined public code repositories for real bug fixes in Chapter 4. Real bug fixes represent real bugs discovered by developers together with a corresponding bug fix. They seem to be a perfect resource for training neural bug detectors. However, because we found that real bugs are scarce in existing open source repositories, we had to mine over 500K projects to obtain a sufficient amount of real bug fixes that can be used for the training of neural bug detectors.

With the help of the mined real bug fixes, we were able to fully evaluate the impact of the training process on the performance of neural bug detectors in Chapter 5. We found that previous neural bug detectors are severely limited by their training process (which confirms Hypothesis 1). The performance of neural bug detectors can be significantly improved by training them on more realistic mutants and on real bug fixes. As a result, we were able to train a neural bug detector that is significantly more effective in finding real bugs.

In Chapter 6, we noticed that neural bug detectors still produce a significant amount of false alarms. We therefore developed a validation approach that uses large language models to validate the output of neural bug detectors. We found that additional file

context which is *not* available to function-level neural bug detectors significantly improves the performance of the validator to reject false alarms. Providing neural bug detectors with file-level access might significantly improve their performance in distinguishing buggy from bug-free code. We see this finding as one of the first evidence that neural bug detectors are limited by their task design (Hypothesis 2). We conclude this thesis by summarizing and discussing the results and by providing an outlook for future work in Chapter 7.

1.4 Publication Details

The ideas presented throughout this thesis are based on several publications. We presented our initial study on the comparison of neural bug detectors and software developers in 2022 at ASE [RHJ+22]. Our idea to use contextual mutation operators for training neural bug detectors, as discussed in Chapter 3, was published in 2022 at ICST [RW22b]. An article on the real bug fix collections which we describe in Chapter 4 was published in the same year at MSR [RW22c]. The study presented in Chapter 5 on the impact of training examples on the performance of neural bug detectors was published at ASE 2023 [RW23]. Our ideas and results presented in Chapter 6 are completely novel and have not yet been published.

2 Background

In this chapter, we provide a brief overview over state-of-the-art methods for automatic bug detection and introduce several background topics that we use throughout this thesis. Additional background will be introduced where necessary in later chapters. In the following, we introduce basic terminology in Section 2.1, we provide an introduction to *neural models of code* in Section 2.2 and we introduce state-of-art methods for *neural bug detection* in Section 2.3.

2.1 Software Bugs

The *IEEE Standard Glossary of Software Engineering Terminology* [C+90] defines the term *software bug* (sometimes also differentiated into *software error* and *fault*) as (1) the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition, (2) an incorrect step, process, or data definition, (3) an incorrect result or (4) a human action that produces an incorrect result. In the context of this thesis[1], we are mostly interested in the way software bugs materialize (Definition 2). In the following, we will use the term *software bug*, *bug* or *error* interchangeably to refer to programming mistakes that lead to *unintended* behavior. In other words, a software bug materializes as a mistake in the implementation where the *actual* runtime behavior deviates from the behavior *intended* by the software developer. The intended behavior can be explicitly defined by *formal specifications* or *test suites*. Software is however rarely accompanied by complete (formal) specifications. We therefore often rely on *implicit* specifications given through *natural hints* in code such as *variable names*, *function signatures* or *code comments*.

[1] The term "bug" is by definition ambiguous and what contributes to a bug is often subjective [WG24]. We still stick to this terminology as it is widely adopted in static bug detection. In our experiments, we often rely on the judgement of the project developers to decide whether something is a bug or not.

2.1.1 Real and Artificial Software Bugs

Throughout this thesis, we often distinguish between *real* software bugs that appear naturally during the coding process and *artificial* bugs that are artificially generated (e.g. for testing purposes [PKZ+19]). Real bugs appear in public repositories in form of bug fixes where the bug is identified by one of the project developers and fixed. Artificial bugs or *mutants*, in contrast, are artificially generated by mutating existing (likely correct) source code with the help of so-called *mutation operators* [PKZ+19, Jus14].

Mutation Operators. Mutation operators can be seen as functions that randomly mutate source code. Assuming that we are given a program P, a mutation operator M is a random function that maps P to a randomly mutated program P_M:

$$P \xrightarrow{M} P_M$$

The goal of the mutation process is to produce buggy variants of P that should be detected by an existing bug detector. Mutation operators have a long history in the context of mutation testing where the goal is the evaluation of test suite [PKZ+19]. In this thesis, we will often borrow mutation operators from mutation testing, but with the goal of training neural bug detectors. In the following, we introduce the mutation operators that are most relevant for our work:

- *Binary Operator Replacement.* We employ a wide range of mutation operators for replacing binary operators. This includes mutation operator for arithmetic operator replacement (AOR), conditional operator replacement (COR), logical operator replacement (LOR), relational operator replacement (ROR) and shift operator replacement (SOR). A binary operator replacement often replaces a binary operator with another random type-compatible operator. For example, the AOR operator and ROR operator might *randomly* mutate the given code snippets as follows:

$$\text{a} + \text{b} \xrightarrow{AOR} \text{a} - \text{b} \qquad\qquad \text{a} <= \text{b} \xrightarrow{ROR} \text{a} > \text{b}$$

- *Unary Operator Insertion, Deletion and Replacement.* A popular mutation strategy is to mutate unary operators randomly by inserting (UOI), deleting (UOD) or replacing unary operators (ORU). Operator replacements are always type-safe. To provide an example, ORU and UOD would replace and delete unary operators as follows:

$$-\text{a} \xrightarrow{ORU} \sim\text{a} \qquad\qquad !\text{a} \xrightarrow{UOD} \text{a}$$

- *Literal Value Replacement.* A literal value replacement (LVR) operator replaces a random literal with another random literal of the same type. For example, LVR replaces numeric and boolean literals as follows:

$$1 \xrightarrow{LVR} 2 \qquad\qquad \text{True} \xrightarrow{LVR} \text{False}$$

In practice, LVR does not replace literals completely randomly. It is more common to replace literals with default values [Jus14] (such as $-1, 0, 1$ for numeric literals).

In addition to the operators mentioned above, we also employ mutation operators less common in mutation testing such as operators for modifying *variable names* or *function calls*.

2.1.2 Categorizing Software Bugs into Patterns

For the evaluation of bug detectors or the design of new bug detection tasks, it is often beneficial to group software bugs into *bug patterns*. In the context of this thesis, we define bug patterns *syntactically* based on the location of the bug and its bug fix. On the highest level, we differentiate between *single token*, *single statement* and *multi statement bugs*. Single token and single statement bugs can be fixed by modifying a single token or a single statement respectively. Multi statement bugs require modification of at least two statements to be fixed. To further categorize single token and single statement bugs, we follow (with a few exceptions) the categorization of "simple stupid bug" (SStuB) patterns proposed by Karampatsis and Sutton [KS20]. In the following, we highlight the bug patterns that are most relevant for our work. Examples are provided in Table 2.1.

- *Change Identifier Bugs*: The developer uses the wrong identifier to access a memory location or to call a function. The bug patterns includes the wrong usage of *variable names*, *parameter names* or *function names*. It is easy for the developer to utilize the wrong identifier with the same type. Similar identifier names (e.g. `patch` and `patches`) that appear in the same code might further contribute to the occurrence of this bug type.

- *Variable Misuse Bugs*: The developer uses the wrong variable name different from the intended one. This bug pattern is not a classical SStuB pattern but still important for this thesis as many neural bug detectors target this bug type [ABK18]. In most instantiations, it is assumed that a variable misuse bug can be fixed with a variable name *defined in the same scope* (e.g. in the function signature or as a local variable).

- *Wrong Function Name Bug (or API Misuse Bug)*: The developer uses a function name different from the intended function name with the same parameter list. While this pattern can also be considered as a part of change identifier bugs, it is often interesting to consider this bug pattern distinctly. The bug pattern is particularly challenging to detect and fix since it requires access to the API defined for the project.

- *Change Operator Bug*: The developer confuses a binary or unary operator. For example, the developer confuses `<=` with `<` in a loop condition resulting in an *off-by-one* error [BSS+20]. Bugs that fall in this bug pattern can be easily generated with mutation operators.

Table 2.1: Examples of Software Bugs found in Python projects [AJFB21]

Example	Description

`# Variable Misuse Bug`
`applied = self.db.applied_patches()`
`for patch in applied:`
`  if patch in patches:`
`  patches.remove(applied)`
	All `applied` patches should be removed from the `patches` list. However, the developer mistakenly tries to remove `applied` instead of a single `patch`. **Fix:** replace `applied` in Line 5 by `patch` defined in Line 3.

`# Wrong Function Name Bug`
`def _size(self):`
`  return len(self.file.readline())`
	The function `_size` computes the size of the stored `self.file`. However, the standard API method `readline` only retrieves the first line of the file. **Fix:** replace `readline` in Line 3 by `readlines`.

`# Change Binary Operator Bug`
`def updateRefractionParameters(self):`
`  ...`
`  if self.checkRefracNoTrack.isChecked():`
`  if self.app.mount.status != 0:`
`  return False`
`  ...`
	The function `updateRefractionParameters` performs an update and returns true if the update was successful. Prior to the update, the function checks if the `mount` is ready and aborts if not. We can conventionally expect that we abort if the status is zero. However, the function checks whether the status is not zero. **Fix:** replace `!=` in Line 5 by `==`.

`# Boolean Literal Bug`
`def isNoneOrEmpty(obj):`
`  if obj is None: return False`
`  if isinstance(obj, list): ...`
	The function `isNoneOrEmpyt` checks if the given `obj` is `None` or empty. However, it returns `False` when the `obj` is `None` . **Fix:** replace `False` in Line 3 by `True`.

- *Change Literal Bug*: The developer uses the wrong numeric or boolean literal. For example, the developer returns accidentally `False` after successfully validating the input to the program.

A full list of all bug patterns together with their description is given in Appendix A.1.

2.2 Neural Models of Code

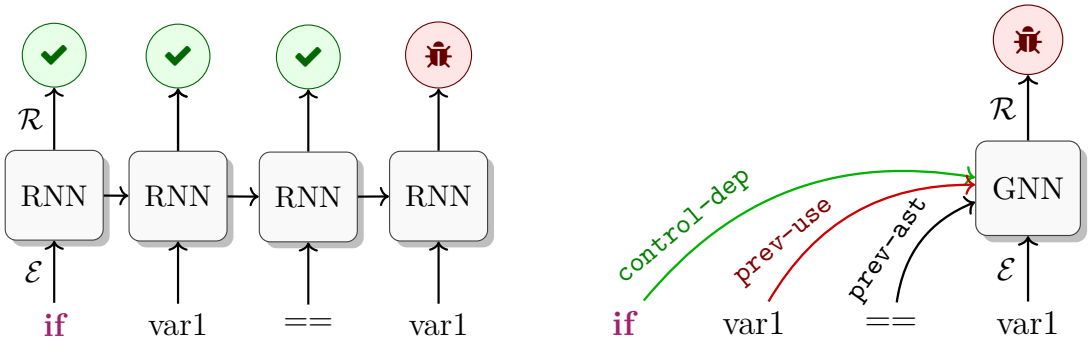
In this section, we discuss *neural models* that have been successfully adopted into the coding domain and which we employ to build neural bug detectors. Neural models are neural network-based architectures that are designed to model some aspect of code. In Section 2.2.1, we review neural architectures that are most relevant for this thesis and, in Section 2.2.2, we discuss important applications. A conceptual overview over the considered architectures used are given in Fig. 2.1 and in Fig. 2.2.

2.2.1 Deep Neural Networks

For the purpose of this work, we view deep neural networks (NN) as a composition of linear transformations $f_1, \ldots, f_L$ interwoven with non-linear transformations σ:

$$NN_\varphi = f_L \circ \sigma \circ \cdots \circ \ \sigma \circ f_1,$$

where $\circ$ denotes the functional composition. Deep neural networks are typically used as *function approximators* with a large number of tunable parameters φ. Neural networks



(a) Recurrent Neural Network (RNN)

(b) Graph Neural Network (GNN)

Figure 2.1: Conceptual Overview of Neural Architectures

can be composed in larger *neural architectures* where the neural network is a basic building block that can be composed with other components. A common configuration for a neural network is the *multi-layered perceptron* (MLP) [R+62]. In an MLP, the linear transformation f_i is defined by:

$$f_i(\mathbf{x}) = \mathbf{W}_i\mathbf{x} + \mathbf{b}_i,$$

where $\mathbf{x} \in \mathbb{R}^n$ is the input to the function, $\mathbf{W}_i \in \mathbb{R}^{m \times n}$ is the weight matrix of the l's layer and $\mathbf{b}_l \in \mathbb{R}^m$ is the bias vector. A common non-linear transformation for ReLU-based MLPs is the ReLU function: $\sigma(\mathbf{x}) = \max(0, \mathbf{x})$. To optimize the parameters φ (which includes $\mathbf{W}_i$ and $\mathbf{b}_i$), we will make use of backpropagation [Wer82] and different variants of stochastic gradient descent (e.g. the Adam optimizer [KB15]) to optimize the neural network NN_φ with respect to some loss function $\mathcal{L}$.

Sequence Representations. To be able to process source code with neural network based architectures, we mainly view source code as a sequence of tokens $T = t_1, t_2, \ldots, t_n$. To obtain the tokenized version of source code, we apply both common language-specific *tokenizers* (e.g. the Python tokenizer) and language-agnostic learned subword tokenizers [SHB16]. In some cases, we also view source code as graphs G. In these cases, the representation is often based on tokens augmented with additional graph relations. Therefore, we mainly focus here on processing token sequences. To be able to process token sequences with neural models, we utilize an *embedding* function $\mathcal{E}$ and a *readout* function $\mathcal{R}$ that maps tokens to embedding vectors $\mathbf{x}_i = \mathcal{E}(t_i)$ and hidden vectors $\mathbf{h}_i$ to usable results $y_i = \mathcal{R}(\mathbf{h}_i)$ respectively. Both $\mathcal{E}$ and $\mathcal{R}$ are often configured as NNs and jointly optimized with the neural architecture to map a sequence of tokens $t_1, t_2, \ldots, t_n$ to a sequence of predictions $y_1, y_2, \ldots, y_n$.

Recurrent Neural Networks. A popular architecture that is specifically designed for sequence based tasks (such as language or code modeling) are *recurrent neural networks* (RNN) [RHW86]. An RNN takes in a sequence of vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ (obtained

through the embedding function) and processes them via a recurrent function:

$$f_{RNN}(\mathbf{x}_i, \mathbf{h}_{i-1}) = \mathbf{h}_i,$$

where $\mathbf{x}_i$ is the i-th input and $\mathbf{h}_{i-1}, \mathbf{h}_i$ are *hidden states*. It is important to note that f_{RNN} is the same neural network in each step and how f_{RNN} reacts to $\mathbf{x}_i$ is mainly a function of $\mathbf{h}_{i-1}$. For processing code, an RNN is typically combined with an *embedding function* $\mathcal{E}$ with $\mathbf{x}_i = \mathcal{E}(t_i)$ and a *readout* function $\mathcal{R}$ with $y_i = \mathcal{R}(\mathbf{h}_i)$. The neural networks f_{RNN}, $\mathcal{E}$ and $\mathcal{R}$ are then jointly optimized to map an input sequence $t_1, t_2, \ldots, t_n$ to predictions $y_1, y_2, \ldots, y_n$. Fig. 2.1a depicts a conceptual overview over the prediction process for the task of bug detection.

Graph Neural Networks. *Graph neural networks* (GNNs) [KW17] assume that code can be represented as a labelled graph $G = (V, E, l_V, l_E)$ with a set of nodes V, edges $E \subseteq V \times V$ and labelling functions $l_V : V \to L$ and $l_E : V \times V \to L$. Nodes are identified by unique identifiers $i \in V$ ranging from 1 to $|V|$ and directed edges $i \to j$ are represented by tuples $(i, j) \in V \times V$. A common choice is to define $V = \{1, 2, \ldots, n\}$ as the set of token indices with $l_V(i) = t_i$. Edges are often defined with the help of common graph representations of code. Popular choices are *abstract syntax trees* [Cho14], *control flow graphs* [All70], *program dependence graphs* [FOW87] and combinations thereof. Now, given a graph $G = (V, E, l_V, l_E)$, a graph neural network often associates nodes with feature vectors $\mathbf{x}_i = \mathcal{E}_V \circ l_V(i)$ via a node embedding function $\mathcal{E}_V$ and edges with feature vectors $\mathbf{x}_{i,j} = \mathcal{E}_E \circ l_E(i, j)$ via an edge embedding function $\mathcal{E}_E$. A GNN then computes hidden representations $\mathbf{h}_i$ and $\mathbf{h}_{i,j}$ for nodes and edges iteratively via a message passing system: Let $\mathbf{h}_i = \mathbf{x}_i$ be the initial hidden representation, the GNN then computes:

$$\begin{aligned} \mathbf{h}_{i,j} &= f_{\text{edge}}(\mathbf{h}_i, \mathbf{h}_j, \mathbf{x}_{i,j}), \\ \mathbf{h'}_i &= f_{\text{node}}\left(\mathbf{h}_i, \sum_{j \in N_i} \mathbf{h}_{j,i}, \mathbf{x}_i\right), \end{aligned}$$

where $N_i = \{j \mid (j, i) \in E\}$ is the neighborhood surrounding i. f_{edge} and f_{node} are typically configured to be small neural networks. Multiple message passing rounds are performed by updating $\mathbf{h}_i$ with $\mathbf{h'}_i$. f_{edge} and f_{node} are not necessarily shared across rounds and it is often more common to define dedicate GNN *layers* for each message passing round. To compute the final prediction, a readout function $\mathcal{R}$ with $y_i = \mathcal{R}(\mathbf{h}_i)$ is used. f_{edge}, f_{node}, $\mathcal{E}_V$, $\mathcal{E}_E$ and $\mathcal{R}$ are often jointly optimized. Fig. 2.1b depicts a conceptual overview of one message passing round together with the prediction process for GNNs.

Transformers. Transformers [VSP+17] represent a class of *non-recurrent* neural networks that process sequences of vectors via an *attention mechanism*. On a very high level, a Transformer can be described as follows: Given a sequence of vectors

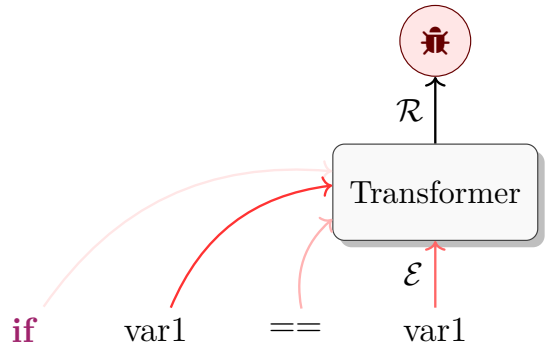


Figure 2.2: High-Level Overview over the Transformer architecture. A Transformer might learn that `if` is less relevant for detecting the variable misuse bug.

$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, a Transformer processes them iteratively to compute hidden representation $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_n$. Let $\mathbf{h}_i = \mathbf{x}_i$ be the initial hidden representation, the Transformer computes:

$$\mathbf{h}_{attn,i} = \mathbf{h}_i + f_O \circ \text{Attn}(\mathbf{h}_i, \{\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h_n}\}),$$
$$\mathbf{h'}_i = \mathbf{h}_i + f_{NN}(\mathbf{h}_{attn,i}),$$

where Attn is an attention mechanism and f_{NN}, f_O are neural networks. The attention mechanism computes an aggregate representation of all hidden representations dependent on $\mathbf{h}_i$:

$$\text{Attn}(\mathbf{h}_i, \{\mathbf{h}_1, \ldots, \mathbf{h_n}\}) = \frac{1}{Z(\mathbf{h}_i)} \sum_{j=1}^{n} \text{sim}(f_Q(\mathbf{h}_i), f_K(\mathbf{h}_j)) f_V(\mathbf{h}_j),$$

where $Z(\mathbf{h}_i) = \sum_{j=1}^{n} \text{sim}(f_Q(\mathbf{h}_i), f_K(\mathbf{h}_j))$, $\text{sim}(\mathbf{x}, \mathbf{y}) = \exp(\mathbf{x} \cdot \mathbf{y} / \sqrt{d})$ with $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ and f_K, f_Q, f_V are neural networks. The Transformer iteratively computes a hidden representation $\mathbf{h}_i$ by setting $\mathbf{h}_i = \mathbf{h'}_i$. Finally, a readout function $\mathcal{R}$ with $y_i = \mathcal{R}(\mathbf{h}_i)$ can be used to compute a result y_i. With the attention mechanism, the Transformer can actively learn which inputs are important to compute the next hidden representation $\mathbf{h}_i$. If an input is unimportant, it will assign a low similarity between $f_Q(\mathbf{h}_i)$ and $f_K(\mathbf{h}_j)$. A higher similarity means that the input is more relevant for computing the hidden representation $\mathbf{h}_i$. Fig. 2.2 provides an example for the attention mechanism used by the Transformer.

Note that we omit a significant amount of technical details for brevity here. Please refer to Vaswani et al. [VSP+17] for a more complete description.

Statistical Optimization. Throughout this work, our goal will often be to fit a statistical model to a given dataset $D = \{(X_i, Y_i)\}_{i=0}^{N}$. X_i are observations which in our case are representations of code snippets. Y_i is the prediction target which could be, for example, whether the given code X_i is buggy or not. It is also feasible that Y_i

is a more complex target such as a text sequence. To fit the statistical model, we often parametrize a neural network NN to approximate a probability distribution $P(Y \mid X)$ which is the probability of observing Y given a code snippet X. A common strategy is to fit the distribution by minimizing the cross-entropy over the given dataset:

$$\mathcal{L} = \mathbb{E}_{(X,Y)\sim D} \left[-\log P_\varphi(Y \mid X) \right],$$

where $\mathbb{E}_{(X,Y)\sim D}$ is the expectation over the dataset and P_φ is a neural network architecture parametrized by φ. As described before, we will use backpropagation and stochastic gradient descent to approximate the optimal parametrization $\varphi^* = \text{argmin}_\varphi \mathcal{L}$.

2.2.2 Important Applications of Code Models

In the following, we review some important applications of neural models of code.

Code Classification. The goal of code classification is to classify a given code snippet X in one of k classes $C^{(1)}, C^{(2)}, \ldots, C^{(k)}$. In a *binary* classification problem, a code classifier has to distinguish between two classes. A *multi-class* classification problem distinguishes between more classes ($k > 2$). To solve a binary or multi-class code classification problem, we often assume that we have access to a dataset $D = \{(X_i, C_i)\}_{i=1}^N$ consisting of programs X_i labelled with a class C_i. Then, we can use a neural model of code to model the probability $P(C \mid X)$ of classifying a code snippet X as belonging to class C. The neural model is trained by minimizing the following cross-entropy loss:

$$\mathcal{L}_{target} = \mathbb{E}_{(X,C)\sim D} \left[-\sum_{i=1}^{k} [\![C = C^{(i)}]\!] \log P_\varphi(C^{(i)} \mid X)) \right],$$

where P_φ is a neural network that approximates the probability distribution $P(C \mid X)$. To then classify a new example X_{new}, we select the class C^* greedily to be the most likely class for X_{new} according to P_φ:

$$C^* = \text{argmax}_C \; P_\varphi(C \mid X_{new})$$

Language Modeling. Language Models (LM) [JM00] are probabilistic models that model the probability of observing a given text, code or token sequence in a given language. Formally, we can represent both text and code as sequence of (program) tokens $T = t_1, t_2, \ldots, t_n$. Then, a language model models the probability of observing T as follows:

$$P(T) = P(t_1, t_2, \ldots, t_n) = \prod_{i=1}^{n} P(t_i \mid T_{<i}),$$

where $P(t_i \mid T_{<i})$ is the probability of seeing token t_i given that we already observed $T_{<i} = t_1, t_2, \ldots, t_{i-1}$. Note that any function that can approximate $P(t_i \mid T_{<i})$, which does not necessitate the use of neural networks, can be seen as a language model.

Neural language models (based on neural networks) however have been empirically shown to be highly effective in language modeling [HD17, KBR+20]. Most notably are *large language models* (LLMs) [BMR+20] which are neural language models with highly parametrized neural networks. To train a language model for code, large corpora of source code $C = \{T_i\}_{i=1}^{N}$ are often used with the goal of minimizing the cross-entropy of the collected code:

$$\mathcal{L}_{LM} = \mathbb{E}_{T=t_1,\ldots,t_n \sim C} \left[-\sum_{i=1}^{n} \log P_\varphi(t_i \mid T_{<i}) \right],$$

where $\mathbb{E}_{T=t_1,\ldots,t_n \sim C}$ is the expectation over the code corpus C and P_φ is a neural network parametrized by φ. Language Models (and especially LLMs) have been largely adopted for the coding domain. Existing approaches have utilized LLMs for code generation [CTJ+21], test case generation [CZN+23] and automatic program repair [JLLT23].

Language models can be used to *generate* code. It is possible to *sample* a code sequence $T \sim P_\varphi(\cdot)$ by iteratively sampling individual tokens (e.g. $t_i \sim P_\varphi(\cdot \mid T_{<i})$). Sometimes we are interested in generating the most likely continuation of $T_{<i}$ which can be done by generating tokens greedily ($t_i = \text{argmax}_t P_\varphi(t \mid T_{<i})$). Generating code based on language models has become its own research field and many methods have been proposed [CTJ+21].

Language models can also be used as a *scoring function* which assigns a probability to every possible code snippet. The score is sometimes referred to as "naturalness" of code [HBS+12] and it suggests how likely it is to observe the given code snippet under the learned model. A code snippet is more *natural* if it is assigned a higher likelihood by a given language model.

Masked Language Modeling. Masked Language Modeling (MLM) [DCLT19] can be seen as variant of language modeling with the goal of modeling the probability:

$$P(t_m, t_{m+1}, \ldots, t_{m+k} \mid t_1, t_2, \ldots, t_{m-1} \texttt{ [M] } t_{m+k+1} \ldots t_n),$$

where `[M]` is a mask in the sequence. These models are typically trained by masking one or multiple tokens in the input. It is also not necessary to mask continuous chunks, but sometimes multiple independent chunks of tokens are masked. The optimization is very similar to language modeling:

$$\mathcal{L}_{MLM} = \mathbb{E}_{T=t_1,\ldots,t_n \sim C} \left[-\log P_\varphi(t_m, \ldots, t_{m+k} \mid t_1 \ldots \texttt{ [M] } \ldots t_n) \right].$$

MLM have been successfully utilized for code problems in the context of *transfer learning* [KB19]: It can be beneficial to first train P_φ with $\mathcal{L}_{MLM}$ resulting in φ_{MLM} and then *fine-tune* the resulting model $P_{\varphi_{MLM}}$ on the target task with $\mathcal{L}_{target}$. Furthermore, and similar to an LM, MLMs can be used to generate likely infilling for the masked token and we can use MLMs to score the "naturalness" of a given infilling [BJT+22].

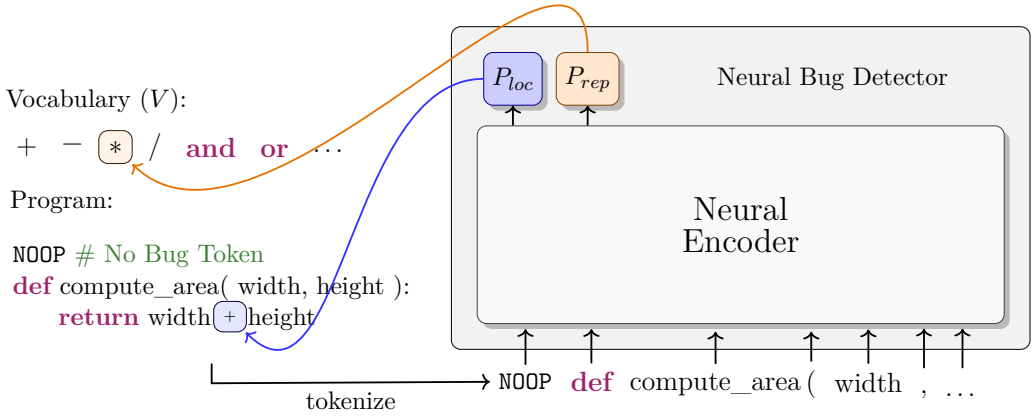


Figure 2.3: Joint Architecture for the Detection, Localization and Repair of Single Token Bugs.

2.3 Neural Bug Detection

Finding and fixing software bugs is a central problem of software development [AL21]. Software developers often spend a significant amount of their time in the debugging process. To mitigate the human effort, many automated approaches based on unit testing, static analysis and formal verification have been developed to assist the developer during debugging. Recently, *static bug detectors* have gained a lot of popularity with bug detection tools such as SpotBugs [HP04], Google's ErrorProne [ASPK12] and Facebook's Infer [CDD+15]. To detect bugs, static bug detectors often implement a wide range of hand-crafted rules for detecting common bug patterns. Then, the implemented rules are checked statically to alert the developer as early as possible about potential mistakes in their implementation.

A key problem of static bug detectors is their low recall paired with a high amount of false alarms. Habib and Pradel [HP18] found that the three static bug detectors SpotBugs, ErrorProne and Infer detect less than 4.5% of all bugs. Karampatsis and Sutton [KS20] found in a subsequent study that SpotBugs identified less than 12% of the bugs found in public repositories while reporting more than 200 million rule violations. Further improving the rules of static bug detectors is however often highly difficult: To increase recall, new rules have to be implemented. However, new rules also risk raising more false alarms. Reducing the false alarms might require to increase the specificity of existing rules which in turn might reduce the number of real bugs detected.

Neural bug detectors [ABK17, PS18, HSS+19, VKM+19, KMBS20, AJFB21, RW22b, HBV22] have been recently introduced as a data-driven alternative to static bug de-

tection. Instead of relying on hand-crafted rules, neural bug detectors employ neural model of code that are specifically trained for the task of bug detection. These neural models are trained on millions of examples of buggy and non-buggy code.

Existing approaches to neural bug detection can be classified into broadly two categories: (1) pure bug detection approaches [ABK17, PS18, BSS+20] and (2) joint approaches [HSS+19, VKM+19, KMBS20, AJFB21, RW22b, HBV22] that combine bug detection with automatic repair. While both approaches have their advantages, we mainly focus in this thesis on joint approaches for bug detection and repair.

2.3.1 Neural Bug Detection and Repair

Joint approaches for neural bug detection and repair commonly address the task of bug detection as three joint *prediction* tasks: Given a potentially buggy program P, the bug detector has to first (1) *classify* the program as buggy or not, (2) *localize* the bug location (if any) and then (3) propose a *repairing* patch. Since patching a buggy program can become arbitrarily complex, most existing works on neural bug detection and repair have focused on the detection and repair of *single token bugs*. Single token bugs are frequent [KS20, ABK17] and can often be easily patched by replacing a single program token. Still, software developers tend to overlook these bugs due to their size, making them a great target for neural bug detection.

Single Token Bug Detection. To better understand the goal of single token bug detection, we can formalize the task as follows: Given a program P tokenized to a sequence of tokens $T = t_1, t_2, \ldots, t_n$, the goal of *single token bug detection* is to detect and repair single token bugs that can be fixed by replacing a token t_l with another token r defined in the same scope ($r \in \{t_1, \ldots, t_n\}$) or in an external vocabulary V (i.e. $r \in V$). To effectively detect and repair single token bugs, neural bug detectors commonly perform the following three tasks: (1) the program represented by T has to be *classified* as buggy, (2) the bug location t_l has to be localized and (3) a *repair* r has to be proposed. In practice, neural bug detectors often model these three tasks jointly as *token replacement* operations. Let T be the buggy version of program and T' the fixed bug-free version, then the neural bug detector is trained to perform the following operations:

$$T \xrightarrow{\text{replace}(t_l, r)} T' \qquad\qquad T' \xrightarrow{\text{noop}()} T'$$

Here, the buggy version T is fixed by replacing t_l with r (replace(t_l, r)) translating it to T'. Since T' is bug-free, a change is not required (noop()).

Probabilistic Model. To be able to learn the detection and repair of single token bugs, a common approach is to model detection, localization and repair as a joint probability distribution over all possible token replacements $\{\langle l, r\rangle \mid t_l \in T \cup \{\texttt{NOOP}\}$ and $r \in T \cup V\}$:

$$P(\langle l, r\rangle \mid T) = P_{loc}(l \mid T) \cdot P_{rep}(r \mid l, T), \tag{2.1}$$

where $P(\langle l, r\rangle \mid T)$ is the probability that T contains a bug that can be fixed by replacing t_l with r (replace(t_l, r)). We include a specific `NOOP` location which indicates that the T is bug-free (noop()). Note that detection, localization and repair is in our case factorized into first localizing the bug location $P_{loc}(l \mid T)$ (including the `NOOP` location for bug-free code) and then finding a patch dependent on the bug location $P_{rep}(r \mid l, T)$. In practice, there also exist other forms of factorization which have been applied successfully for neural bug detection. For example, Vasic et al. [VKM+19] modeled the probability distribution independently ($P(\langle l, r\rangle \mid T) \approx P_{loc}(l \mid T) \cdot P_{rep}(r \mid T)$). He et al. [HBV22] proposed a task hierarchy by modeling the detection, localization and repair independently. If not stated otherwise, we mainly stick to the factorization shown in Eq. (2.1).

Inference. Given the distribution $P(\langle l, r\rangle \mid T)$, single token bugs can be easily localized and repaired, e.g. by computing the most likely repair for a given task as follows:

$$\text{replace}(t_{l'}, r') \text{ with } \langle l', r'\rangle = \text{argmax}_{l,r} P(\langle l, r\rangle \mid T)$$

In practice, $P(\langle l, r\rangle \mid T)$ is unknown. Therefore, our goal is to approximate $P(\langle l, r\rangle \mid T)$ by approximating $P_{loc}(l \mid T)$ and $P_{rep}(r \mid l, T)$ with neural network-based architectures.

2.3.2 Neural Architecture for Bug Detection and Repair

In this section, we review a common neural architecture often used by existing works[2] for approximating $P(\langle l, r\rangle \mid T)$. A conceptual overview is provided in Fig. 2.3. The architecture mainly consists of two *pointer networks* [VFJ15] for localizing bugs in the input program and for identifying repairs found in the same program or coming from an external vocabulary. Both pointer networks share a common *neural encoder* which computes a vector representation of the input program tokens. The architecture is modular and it is possible to swap out the neural encoder with any neural model that is able to process code. Existing works have exploited this modularity to evaluate various neural encoders based on RNNs [VKM+19], GNNs [AJFB21], Transformers [KMBS20] and hybrid models [HSS+19].

Pointer Networks for Bug Localization. Pointer networks model the probability of selecting (*pointing to*) certain elements in the neural network's input. For a given program P that is tokenized to $T = t_1, \ldots, t_n$, a pointer network can model the probability of selecting t_i from the input token sequence. To be able to process T with a pointer network, the token sequence has to be first encoded by a neural encoder that maps $t_1, \ldots, t_n$ to a sequence of hidden vector representations $\mathbf{h_1}, \ldots, \mathbf{h_n} \in \mathbb{R}^h$. Then, the pointer network models the probability of selecting t_i as a *softmax* distribution over

[2] The neural architectures slightly differ between different research projects. We review in this section an architecture that summarizes most of the advances made by previous works including our work in [RW23].

all hidden vectors:

$$P_{ptr}(t_i \mid T) = \frac{exp(\psi(t_i))}{\sum_{j=1}^{n} exp(\psi(t_j))},$$

where $\psi(t_i)$ is a learnable scoring function (e.g. $\psi(t_i) = \mathbf{W}^\intercal \mathbf{h}_i$ with $\mathbf{W} \in \mathbb{R}^{1 \times h}$). $\psi(t_i)$ computes a scalar for each token t_i that indicates how likely t_i is selected. If $\psi(t_i)$ is higher than any other $\psi(t_j)$, t_i is more likely to be selected. There exists several variants of pointer networks [VKM+19, SLM17, GLLL16] that apply transformations on the hidden vectors to incorporate more context or that mask out certain elements from the input. In the end, most existing pointer techniques use the same pointing mechanism.

Copy Mechanism for Bug Repair. A *copy mechanism* [GLLL16] can be used to copy a certain word or token from the input. Let $T = t_1, \ldots, t_n$ be the input token sequence and $\mathcal{T} = \{t_1, \ldots, t_n\}$ the set of *unique* tokens in the input, the goal of the copy mechanism is to model the probability of copying a token $y \in \mathcal{T}$. We can model the copy probability with pointer networks. However, since y might occur multiple times in the input, we need to aggregate the probabilities of selecting a token t_i with $y = t_i$:

$$P_{cpy}(y \mid T) = \sum_{j:t_j=y} P_{ptr}(t_j \mid T),$$

The copy mechanism on its own can only model the probability of copying from the input. In some cases, we might however be interested in modeling the probability of selecting a token y that might not occur in the input. For this task, we can use a mechanism similar to CopyNet [GLLL16]: Assume that we have an external vocabulary $V = \{v_1, \ldots, v_m\}$ in addition to the input program T, the probability of predicting $y \in T \cup V$ can be modeled as follows:

$$P_{vcpy}(y \mid T) = P_V(y \mid T) + P_{c\hat{p}y}(y \mid T),$$

where $P_V(y \mid T)$ and $P_{c\hat{p}y}(y \mid T)$ are defined as:

$$P_V(y \mid T) = \begin{cases} \frac{1}{Z}\exp(\psi_g(y)) & \text{if } y \in V \\ 0 & \text{else} \end{cases}$$

$$P_V(y \mid T) = \begin{cases} \sum_{j:t_j=y} \frac{1}{Z}\exp(\psi_c(t_j)) & \text{if } y \in T \\ 0 & \text{else} \end{cases}$$

The functions $\psi_g(y)$ and $\psi_c(t_j)$ are scoring functions for selecting $y \in V$ or $y = t_j \in T$ respectively. The term $Z = \sum_{v \in V} \exp(\psi_g(y)) + \sum_{t_j \in T} \exp(\psi_c(t_j))$ is a shared normalization term. In practice, we can use this copy mechanism directly to model $P_{rep}(r \mid l, T)$ where $r \in V$ or $r \in T$. However, the computation of the repair would be independent of the bug location (except for information provided through the shared encoder).

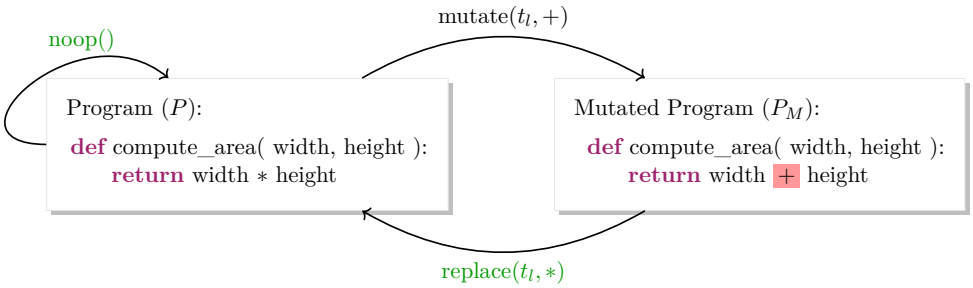


Figure 2.4: Generating Artificial Training Data for Neural Bug Detection. Green operations are operations used for training neural bug detectors. The operation noop indicates that no change is required. The operation replace(t_l, r^{-1}) inverts the mutation process.

To incorporate the bug location for predicting the repair, we can modify the scoring functions as follows:

$$\psi_g(y \mid t_l) = \mathbf{W}_y^\intercal \mathbf{h}_l \qquad \text{and} \qquad \psi_c(t_j \mid t_l) = \mathbf{h}_l \mathbf{W} \mathbf{h}_j,$$

where $\mathbf{W}_y \in \mathbb{R}^{1\times h}$ is a linear projection specific to y and ψ_c is a bilinear projection dependent on t_j and t_l with projection matrix $\mathbf{W} \in \mathbb{R}^{h\times h}$. We define $P_{vcpy}(r \mid l, T)$ to be $P_{vcpy}(r \mid T)$ using the modified scoring functions during its computation.

Training. Given a dataset $D = \{T_i, \langle l_i, r_i\rangle\}_{i=1}^N$ of token sequences annotated with bug locations and repairs, a neural bug detector is trained to minimize:

$$\mathcal{L}_{loc+rep} = \mathbb{E}_{T,\langle l,r\rangle\sim D}\left[-\log P_\varphi(\langle l, r\rangle \mid T)\right],$$

where $P_\varphi(\langle l, r\rangle \mid T) = P_{ptr}(t_l \mid T) \cdot P_{vcpy}(r \mid l, T)$ is a neural bug detector parametrized by φ.

2.3.3 Generating Training Data by Mutating Programs

Obtaining a dataset D of token sequences annotated with bug locations and repairs at a sufficient scale is often a challenging problem. Real bugs are scarce in open source projects [KS20] and mining real bugs from commits can only find bugs that made their way into the project. As a result, many existing works have employed *mutants* for training neural bug detectors. Mutants can be easily obtained at large scale by mutating existing source code.

Training with Mutants. Given a program P tokenized to $T = t_1, \ldots, t_n$, the goal of training with mutants is to learn to invert the mutation process:

$$T \xrightarrow{\text{mutate}(t_l, r)} T_M \xrightarrow{\text{replace}(t_l, r^{-1})} T,$$

where T_M is the mutated token sequence which can be obtained by mutating T with mutate(t_l, r). The neural bug detector is trained to repair mutants ($T_M \xrightarrow{\text{replace}(t_l, r^{-1})} T$) and detect the absence of mutations ($T \xrightarrow{\text{noop}()} T$). Fig. 2.4 depicts an example of the mutation process used for generating training data. Ultimately, given a corpus of tokenized code $C = \{T_1, \ldots, T_N\}$, we generate training examples of mutated code $(T_M, \langle l, r^{-1} \rangle) \in D$ by mutating existing source code ($T \xrightarrow{\text{mutate}(t_l, r)} T_M$) and we generate examples of unmutated code $(T, \langle 0, r \rangle) \in D$ with $t_0 =$ `NOOP`. In practice, the mutation operator is often applied multiple times for generating up to k mutants. In this case, a common strategy is to balance the dataset by introducing k copies of the unmutated code (where k is the number of generated mutants).

3 Mutations for Neural Bug Detection

As we have seen in the last chapter, existing neural bug detectors are often trained on artificially generated mutants. These mutants are generated by *randomly* mutating existing source code. Because of the random process, we hypothesize in this chapter that mutants used for training of neural bug detectors are often not representative for real bugs. To test this hypothesis, we propose a *contextual* mutation operator that is able to generate more realistic mutants based on the surrounding context. We show that the contextual mutation operator is significantly more effective in generating realistic bugs. With the help of the new mutator, we are able to train neural bug detectors that are significantly more effective in detecting real bugs.

3.1 Motivation

Artificial bugs, in the form of code mutants, have long been studied in the field of software testing, especially within the context of *mutation testing* [PKZ+19, Jus14]. The key idea is that a bug detector *sensitive* enough to detect code mutants is also sensitive enough to detect more complex real bugs. This phenomenon, also called *coupling effect* [Off92, JJI+14], was empirically observed in multiple studies and the *mutation score*, a metric that measures how many mutants can be detected, has since then become a standard metric to measure the quality of test suites. Code mutants have also been used to create *benchmarks* for bug finding tools [DHK+16, RPDH18]. Here, a tool is identified as more effective if it catches more mutants.

Recent studies [HSS+19, VKM+19] on neural bug detectors have observed a similar coupling effect: Neural bug detectors that are more effective in detecting mutants generally tend to be more effective in finding real bugs. However, this effect is limited. In comparison to their performance on artificial benchmarks, neural bug detectors often significantly underperform when evaluated on real bugs [VKM+19, HSS+19, AJFB21].

In this chapter and in Richter and Wehrheim [RW22b], we hypothesize that this gap

(a) Source

```
if (dataset == null) return result;
```

(b) Loose mutant

```
if (dataset / null) return result;
```

(c) Strict mutant

```
if (dataset <= null) return result;
```

(d) Contextual mutant

```
if (dataset != null) return result;
```

Figure 3.1: Code snippet taken from Defects4J/Chart#1.

in performance is due to a lack of realistic bugs in the training set. Contrary to other bug-related techniques such as test suites or static bug finders, neural bug detectors fit the *distribution* of bugs seen during training. Therefore, if the bugs in the training set do not closely resemble real programming mistakes, a neural bug detector will not be able to perform well on real bugs.

An alternative approach which would likely mitigate this problem would be to directly train on real bugs obtained from mining public repositories [KS20, JJE14, SLL+18, GVS+19, WSL+20]. However, since bugs are rare and mining can only find the bugs that have found their way into the repositories, existing collections (such as Defects4J [JJE14], bugs.jar [SLL+18] in Java, BugsJS [GVS+19] in JavaScript and BugsInPy [WSL+20] in Python) often contain only a few hundred examples, which are too few for proper generalization.

To still be able to evaluate the impact of more realistic training examples on the training of neural bug detectors, we introduce in this chapter a novel *contextual mutation operator*. The key idea is to employ a masked language model (MLM) [DCLT19] to evaluate – based on the surrounding context – how likely a mutant would appear in a real project. The MLM employed in our work is trained on millions of programs (including buggy ones) with the objective of predicting the replaced token at a masked location. To generate code mutants, our mutation operator first masks out a random mutation location and then samples a single token mutation according to the distribution computed by the MLM. As a result, our contextual mutation operator can be used to seed various single token bugs including *variable misuse bugs* [ABK17] and *binary operator bugs* [PS18] which are often studied in the context of neural bug detection.

Example. To further motivate the use of contextual mutation operators for the training of neural bug detectors, we consider the example code snippets depicted in Fig. 3.1. The example depicts one line of a code taken from the Defects4J [JJE14] benchmark and three variants that are obtained by applying different form of mutations. The first *loose* mutant is generated by arbitrarily replacing a binary operator `==` with another binary operator `/`. Mutations of this type without any restriction are often employed to gen-

erate training examples for neural bug detectors [PS18, VKM+19, BSS+20, HSS+19]. The second *strict* mutant is generated by following a stricter mutation process. Here, we replace a relational operator `==` with another relational operator `<=`. While this stricter form of mutation is more common in mutation testing, it still generates mutants solely on non-contextual information, thereby ignoring that `==` is used for a comparison with a `null` literal. Our contextual mutation operator takes the context of the source code into consideration. By having learned that `==` and `!=` are common in this context, it injects `!=` as a replacement for `==`. Such a contextual mutation is often more realistic than the loose or strict one, and for this example actually has been the original bug which was present in the code snippet.

3.2 Mutation Operators

We start with a more general view on mutation operators commonly used in the training of neural bug detectors. The following definitions will then help us for the description of our contextual mutation operator in Section 3.3.

Single Token Mutations. As our goal is the generation of training data for neural bug detectors, we focus on the generation of single token bugs. To generate single token bugs, we mutate existing code by introducing *single token mutations*: Given a program P tokenized[1] to $T = t_1, \ldots, t_n$, a single token mutation is introduced by replacing a single token t_l with another token r. The token r can be a token defined in the same scope ($r \in \{t_1, \ldots, t_n\}$) or coming from an external vocabulary ($r \in V$). Single token mutations are randomly introduced via a *mutation operator* $\mathcal{M}$. This is typically done by sampling from a probability distribution[2] $P_{\mathcal{M}}$ over all possible single token mutations $\{\langle l, r\rangle \mid t_l \in T, r \in T \cup V \text{ and } t_l \neq r\}$:

$$\text{mutate}(t_l, r) \text{ with } \langle l, r\rangle \sim P_{\mathcal{M}}(\cdot \mid T),$$

which is then applied to T to generate a *mutant* T_M ($T \xrightarrow{\text{mutate}(t_l,r)} T_M$).

Mutation Operator Types. In this work, we consider different types of mutation operators. The operator type is determined by the set of mutations $M_T = \{\langle l, r\rangle \mid P_{\mathcal{M}}(\langle l, r\rangle \mid T) > 0\}$ that a mutation operator $\mathcal{M}$ can generate for a given program T. Let Op_T be the set of valid mutations for a specific mutation operator type Op, then a mutation operator $\mathcal{M}$ is of type Op iff $M_T \subseteq Op_T$ for all programs T. For example, a given mutation operator $\mathcal{M}$ is a binary operator replacement (BOR) mutation operator

[1] For simplicity, we will use in the rest of this chapter program, implementation and their tokenized version interchangeably. Formally, we use T to denote the tokenized version of program or implementation.

[2] We mainly focus here on single token replacements such AOR, COR, and LVR. Insertions and deletions are therefore also handled as token replacements, e.g. by replacing a token with an empty token (deletion) or by replacing a token with a prefixed token (insertion).

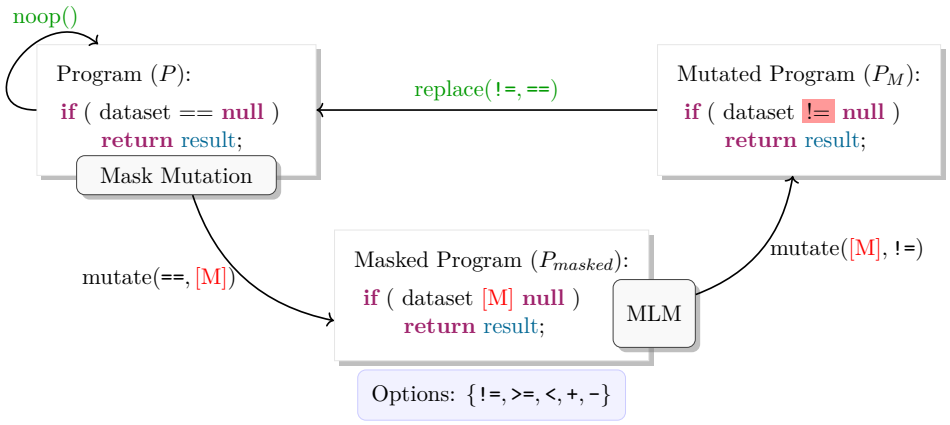


Figure 3.2: Overview over our contextual mutation process

iff $M_T \subseteq BOR_T$ for all T and BOR_T is defined as follows:

$$BOR_T = \{\langle l, r\rangle \mid t_l, r \in BOP, t_l \neq r\} \text{ with } BOP = \{\texttt{==}, \texttt{!=}, \texttt{+}, \texttt{-}, \dots\}$$

Operator Strictness. In our initial example, we distinguished between two different levels of *operator strictness*: loose and strict. To formalize this categorization, we define the operator strictness of a mutation operator $\mathcal{M}$ as the way in which the operator restricts the mutation process. A mutation operator $\mathcal{M}_1$ is *stricter* than another operator $\mathcal{M}_2$ for a given program T iff $\mathcal{M}_1$ considers a smaller set of mutations than $\mathcal{M}_2$, i.e. $M_1 \subseteq M_2$ where M_1, M_2 are mutations generated for T by $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively. Furthermore, an operator $\mathcal{M}_1$ is generally stricter than another operator $\mathcal{M}_2$ if it is stricter for all possible programs T. Now, let Op_T be the set of valid mutations for a specific operator type Op, then we say that a mutation operator $\mathcal{M}$ is a *loose* mutation operator of type Op iff $M_T = Op_T$ for all T. A *strict* mutation operator of type Op is a mutation operator $\mathcal{M}$ of type Op that further restricts the mutation process, i.e. $M_T \subset Op_T$ for all T. Note that a strict mutation operator $\mathcal{M}_1$ is always stricter than a loose mutation operator $\mathcal{M}_2$ of the same Op.

3.3 Contextual Mutations

In this section, we introduce our *contextual mutation operator*. The operator consists of two components: (1) a *mask mutation operator* that both masks out a token from the input and computes the set of potential mutation candidates and (2) a *masked language model* (MLM) that selects mutation candidates according to the given context. An overview of our method is provided in Fig. 3.2. Here, the masked mutation operator first selects a random location to be mutated, e.g. the `==` operator in our example. Then, it generates a masked program by masking out the mutation location. The masked mutation operator also computes a set of mutation *options* which represent

all valid replacements of the mask that produce valid mutations, e.g. BOR mutations. Finally, the MLM assigns a probability to each mutation option which we use to sample a contextual mutation. In the following, we describe our method in more detail. In Section 3.3.2, we start with a formal description of a contextual mutation operator. The mask mutation operator is introduced in Section 3.3.2. In Section 3.3.3, we discuss how (masked) language models can be utilized to select mutation candidates.

3.3.1 Contextual Mutation Operator

Our goal is a *contextual mutation operator* that generates mutations based on the given *mutation context*. A mutation context is defined by the implementation T that should be mutated together with the mutation location l that describes where the code is mutated. Based on this definition, we characterize a contextual mutation operator as follows:

Definition 3.1 (Contextual Mutation Operator)**.** *A mutation operator $\mathcal{M}$ is a contextual mutation operator iff the sampling distribution $P_{\mathcal{M}}$ is different for at least one pair of two different contexts (l_1, T) and (l_2, T') with $(l_1, T) \neq (l_2, T')$ and $t_{l_1} = t'_{l_2}$. In other words, there exists a replacement r such that:*

$$P_{\mathcal{M}}(\langle l_1, r\rangle \mid T) \neq P_{\mathcal{M}}(\langle l_2, r\rangle \mid T')$$

Note that our definition requires that a contextual mutation operator considers the context *surrounding* the mutation location. If a mutation operator only considers the token to be mutated, e.g. the distribution is only different because $t_{l_1} \neq t'_{l_2}$, then this mutation operator is not contextual.

Example. To provide a concrete example, let us slightly adapt the code snippet from Fig. 3.1 to include another mutation location:

```
if ( dataset ==1 null || dataset.size() ==2 0 ) return result;
```

In this example, there exists three locations to perform a binary operator replacement mutation. To highlight the impact of the mutation context, we focus on the two mutation locations that modify the `==`-operator which are highlighted in this example. Since we are mutating the same operator, how we mutate the operator to achieve a realistic mutant mainly depends on the surrounding context. The first operator $==_1$ is used for a comparison with a `null` literal. As `==` and `!=` are the only valid operators for `null` comparisons, a developer might confuse these two operators. Confusing `==` with another operator is less likely for an experienced developer. In contrast, the second operator $==_2$ compares with an integer literal. In this context, replacing the $==_2$-operator with operators like `>`, `>=` or `!=` are all equally valid replacements.

Decomposition. To derive a contextual mutation operator $\mathcal{M}_{ctx}$, we decompose the

sampling distribution $P_{\mathcal{M}_{ctx}}$:

$$P_{\mathcal{M}_{ctx}}(\langle l, r\rangle \mid T) = P_{\mathcal{M}_{ctx}}(l \mid T) \cdot P_{\mathcal{M}_{ctx}}(r \mid l, T)$$

In the following, we model $P_{\mathcal{M}_{ctx}}(l \mid T)$ indirectly with a mask mutation operator that randomly masks a mutation location l in T. To obtain a contextual mutation operator, we employ a language model to approximate $P_{\mathcal{M}_{ctx}}(r \mid l, T)$ dependent on the context (l, T).

3.3.2 Generating Mutation Candidates with a Mask Mutator

For selecting a mutation context together with a set of valid mutations, we employ a *mask mutation operator*. The mask mutation operator randomly samples a location to be mutated and then masks the token in the sampled location with a special mask token ([M]). The result of this masking mutation is a masked program T_{masked} and a set of mutation candidates that lead to valid mutations of the original program.

Mask Mutation Operator. Let M_T be a set of *valid* mutation candidates for a program T and $Loc_T = \{l \mid \langle l, r\rangle \in M_T\}$ be the set of mutation locations, then we define the mask mutation operator $\mathcal{M}_{mask}$ based on the following sampling distribution $P_{\mathcal{M}_{mask}}$:

$$P_{\mathcal{M}_{mask}}(\langle l, \texttt{[M]}\rangle \mid T) = \frac{1}{|Loc_T|}$$

which we use to mask out a certain token in T at location $l \in Loc_T$. To generate a masked program T_{masked}, we sample a random replacement $\langle l, \texttt{[M]}\rangle \sim P_{\mathcal{M}_{mask}}(\cdot \mid T)$ and replace the token t_l with [M] (mutate$(t_l, \texttt{[M]})$). Now, let $Rep_{t_l} = \{r \mid t_l \in T, \langle l, r\rangle \in M_T\}$ be the set of valid replacements for t_l, we define the set of valid mutations $M_{T_{masked}}$ for T_{masked} as follows:

$$M_{T_{masked}} = \{\langle l, r\rangle \mid t_l = \texttt{[M]}, r \in Rep_{t_l}\}$$

Example. To provide a concrete example how our mask mutation operator works, we consider again the example from Section 3.3.1 with the goal of injecting a binary operator mutation. Given the set of valid mutations $M_T = \{\langle l, r\rangle \mid t_l, r \in \{\texttt{==}, \texttt{!=}, \texttt{+}, \texttt{-}, \dots\}, t_l \neq r\}$ for a BOR mutation operator, the mask mutation operator randomly selects a location in $\{l \mid \langle l, r\rangle \in M_T\}$ and replaces it with a special mask token, e.g.:

```
if ( dataset [M] null || dataset.size() == 0 ) return result;
```

In addition, the mask mutation operator generates a new set of valid mutations $M_{T_{masked}} = \{\langle l, r\rangle \mid t_l = \texttt{[M]}, r \in BOP \setminus \{\texttt{==}\}\}$, i.e. all binary operators excluding the equality operator.

3.3.3 Contextual Mutant Selection with Language Models

Before we describe the integration of language models in the mutation process, we want to again motivate why language models are feasible choice for generating realistic mutations with an example:

```
for ( int i = 0; i [M] length; i++ )
```

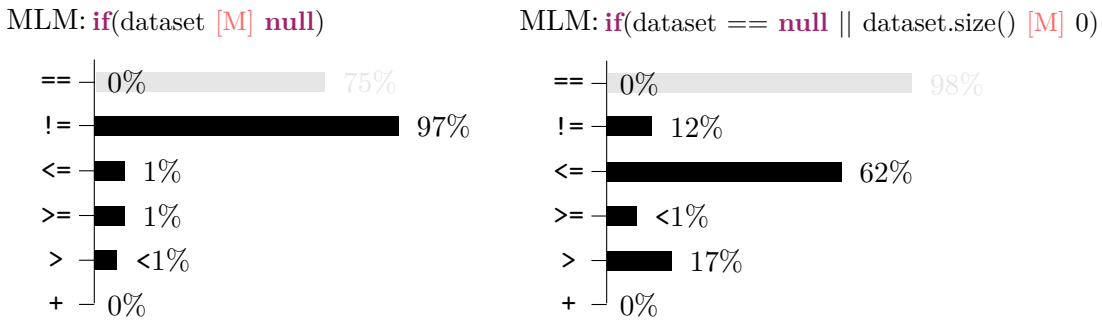
Let us assume for a moment that we want to find the most likely or correct replacement of the mask in the given context of the program snippet. Even though only the loop head is provided, an experienced developer can still make an educated guess: As the loop starts at index 0 and increments the counter after each iteration, the use of a less (`<`) or less equal (`<=`) operator is most likely. Other operators {`==`, `!=`, `>`, `>=`} are unlikely since their use is either uncommon in this context, would terminate the loop from the beginning or yield an infinite loop. A language model would come to the same conclusion, simply because it has learned that the less operators are more frequent in this context. For our contextual mutation operator, we exploit the same judgement for mutation. For example, assuming that the original operator was in fact a less operator, selecting the next likely operator (`<=`) according to a language model would result into an *off-by-one* error [BSS+20]. Because we observed that language models frequently rank realistic mutants highest, we propose to exploit language models as an automatic way for generating contextual mutations.

Masked Language Models. Masked Language Models (MLM) [DCLT19] model the probability of masked out tokens given the rest of the program as context. In the context of this work, we employ MLMs trained for single token replacement like BERT [DCLT19] and CodeBERT [FGT+20]. Given a program $T = t_0 \dots t_n$, these MLMs are trained to predict the probability:

$$P_{\mathbf{LM}}(t_m \mid T_{masked}),$$

where $T_{masked} = t_0 \dots t_{m-1}[\mathrm{M}]t_{m+1} \dots t_n$ is equal to T at all locations $i \neq m$ and uses a special mask token [M] for t_m. In our case, we generate T_{masked} with the help of our mask mutation operator. To our advantage, the probability distribution is defined over a large set of program tokens. Hence, it is possible to apply MLMs for all kinds of single token mutations including mutations of binary operators but also mutations of identifiers variable usages and function calls. These mutations are traditionally difficult to address with classical mutation operators.

Mutation re-weighting. The goal of an MLM is to reproduce the original masked out token given its surrounding context. To force the MLM to generate mutants of T instead of reproducing the original implementation, we have to restrict the probability distribution $P_{\mathbf{LM}}$ to the set of valid mutants $M_{T_{masked}}$. For this, we employ the following



(a) Comparison with `null`

(b) Comparison with integer

Figure 3.3: Re-weighted probability distribution $P_{\mathbf{LM}}$ for two different contexts.

re-weighting method for all $t \in M_{T_{masked}}$:

$$P_{\mathbf{LM}}(t \mid t_m, T) = \frac{P_{\mathbf{LM}}(t \mid t_m = [\mathrm{M}], T_{masked})}{\sum_{v \in M_{T_{masked}}} P_{\mathbf{LM}}(v \mid t_m = [\mathrm{M}], T_{masked})}$$

The probability of all other tokens in V are set to zero. Assuming that $P_{\mathbf{LM}}$ is a softmax distribution, re-weighting is equivalent to reducing the vocabulary V to $M_{T_{masked}}$. An example for our re-weighting method is shown in Fig. 3.3a. Here, the MLM originally assigned a high probability to `==`. By re-weighting, the likelihood of sampling a realistic mutant (`!=`) significantly increases.

Sampling distribution. By combining the mask mutation operator with the MLM-based mutation re-weighting, we can construct $P_{\mathcal{M}_{ctx}}$ of our contextual mutation operator $\mathcal{M}_{ctx}$:

$$P_{\mathcal{M}_{ctx}}(\langle l, r\rangle \mid T) = P_{\mathcal{M}_{mask}}(\langle l, [\mathrm{M}]\rangle \mid T) \cdot P_{\mathbf{LM}}(r \mid t_l, T)$$

Our construction allows us to construct contextual mutation operators for mutation operator type Op considered in this chapter. In addition, it is also possible to derive a contextual mutation operator based on any existing single token mutation operator $\mathcal{M}$ (by computing the mutation set M_T). In our experiments, we utilize this property to construct contextual mutation operators for several bug types.

Example. Whether $\mathcal{M}_{ctx}$ is in fact a contextual mutation operator highly depends on the distribution $P_{\mathbf{LM}}$. To provide an example for an MLM that can be used to construct contextual mutation operators, we consider CodeBERT [FGT+20] and evaluate it for our example in Fig. 3.1 and in Section 3.3.1. In Fig. 3.3, the probability distribution $P_{\mathbf{LM}}$ for two contexts are depicted. In the first case in Fig. 3.3a, the MLM-based contextual mutation assigns a high probability to replacing `==` with `!=`. Since we are replacing in Fig. 3.3b a comparison operator with an integer, the contextual mutation operator assigns a higher probability to operators that are common for integer

Table 3.1: Syntactic roles distinguished by the syntactic tagger. Colors are used for visualizing the roles of tokens throughout the paper.

Type	Description
`Syntax`	Syntactic elements such as brackets
`Keyword`	Language keywords
`Func_Def`	Name of function in signature
`Func_Call`	Name of function in function call
`Var_Def`	Name of variable during definition
`Var_Use`	Name of variable in a loading context
`Unary`	Unary operator, e.g. boolean negation
`Binary`	Binary operator, e.g. float division
`Assign`	Assignment operator
`Type`	Name of type used in typed languages
`Attr`	Object attribute
`String`	String literal
`Number`	Number literal
`Name`	Name which is no variable, function or attribute

comparisons.

3.3.4 Implementation

To evaluate the impact of different mutation operators on the training of neural bug detectors, we implemented all employed mutation operators in a common mutation framework. To simplify the identification of mutation candidates, we developed a *syntactic tagger*. The tagger analyzes the structure of the program in form of the abstract syntax tree (AST) and assigns each individual token a syntactic role. We implemented the syntactic tagger on top of commonly available parsers[3] for AST parsing. Formally, the syntactic tagger computes a function $roles : \{t_1, \ldots, t_n\} \to S$ that assigns each token in $T = t_1, \ldots, t_n$ a role in S. In total, our syntactic tagger distinguish between 14 different roles (see Table 3.1).

To define a mutation operator type Op with the help of the syntactic tagger, it is sufficient to define a role $R \in S$ (e.g. `Binary`) and a set of replacement candidates C_T (e.g. the set of binary operators). Potential mutation candidates are identified by scanning the given program T for tokens with role R and by computing all valid replacements of the found token based on C_T. The contextual mutant selection is built upon the official BERT implementation from the `transformer` library [WDS+19]. Since BERT uses a subtoken vocabulary, it can happen that a token is split into multiple subtokens. In this case, we average the subtoken log probabilities to obtain the sampling probability $P_{\mathbf{LM}}$ of a single token. In our experiments, we use CodeBERT which is a variant of BERT specifically trained for code.

[3] e.g `javalang` for Java, `libcst` for Python and `esprima` for JavaScript

```
if(target.length == 0) {
  // Operates on elements in target
}
```

(a) Binary operator replacement

```
Object o1 = stack.pop();
Object o2 = stack.pop();
if(o1 instanceof Integer){...}
if(o1 instanceof Integer){...}
```

(b) Variable misuse

```
public void stopTest() {
  //...
  mRMClient.start();
 }
```

(c) Function misuse

Figure 3.4: Examples of all studied bug types taken from our real world benchmark. Reformatted and abbreviated for visualization.

3.4 Evaluation

In our evaluation, we investigate the impact of mutant realism on the training of neural bug detectors. For this, we compare the impact of three types of mutation operators:

$$\text{loose} \quad \sqsupseteq \quad \text{strict} \quad \sqsupseteq \quad \text{contextual}$$

Our goal is to show that the mutants generated by loose, strict and contextual (strict) mutation operators are increasingly realistic. Then, we investigate the impact of these mutation operators on the training of neural bug detectors. In the process, the following research question have guided our evaluation:

RQ1 Are contextual mutants more realistic than loose and strict mutants?

RQ2 Can training on realistic mutants improve the effectiveness of neural bug detectors in Java?

RQ3 Does the effect of more realistic mutants transfer to other programming languages and other bug types?

In RQ1 and RQ2 we focus our evaluation on real bugs found in public Java repositories. To explore the effect of our mutation operator on neural bug detectors in other languages, we evaluate neural bug detectors for Python and JavaScript bugs in RQ3.

3.4.1 Evaluation Tasks

To answer our research questions, we evaluate mutation operators on two types of tasks: (1) the reproduction of real bugs and (2) the generation of training data for neural bug detectors. All evaluation tasks, datasets and evaluated bug types together with dataset statistics are summarized in Table 3.2.

Reproduction. To assess the ability of mutation operators to generate realistic bugs, we evaluate them on the task of reproduction. The goal of the reproduction task is to *reproduce* realistic bugs found in real world projects. In **RQ1**, we focus on the reproduction of three common types of Java bugs including binary operator replacement (BOR) bugs [PS18], variable misuse (VarMisuse) bugs [ABK17] and function misuses (FuncMisuse) bugs [KBR+20]. Examples of these bug types found in real projects are shown in Fig. 3.4. To evaluate the ability of the mutation operators to reproduce

Table 3.2: Overview for all evaluation tasks, datasets and bug types individually for each research question. For brevity, we omit the dataset statistics for individual mutator types in the same bug category, since this is shared.

Task	Datasets	Lang	Bug Type	Train	Test
Reproduction (RQ1)	ManySStuBs4J [KBR+20]	Java	BOR	-	1888
			VarMisuse	-	1898
			FuncMisuse	-	2436
Detect & Repair (RQ2)			BOR	837K	1888
- Train	CodeSearchNet [HWG+19]	Java	VarMisuse	2M	1898
- Test	ManySStubs4J [KBR+20]		FuncMisuse	2M	2436
Detect & Repair (RQ3)					
- Train	ETH Py150 [RBV16]	Python	VarMisuse	2M	882
- Test	PySStuBs [KPBH21]				
Detect & Repair (RQ3)					
- Train	ETH Js150 [RBV16]	JavaScript	BOp	1.16M	377
- Test	SemSeed-Eval [PP21]				

these bug types, we derive a real world benchmark based on bug fixes collected in the ManySStuBs4J dataset [KBR+20].

To measure the effectiveness of a mutation operator on the reproduction task, we employ the *inverse Brier score* [Bri50]. The inverse Brier score measures the likelihood of a mutation operator to reintroduce a real bug into a repaired program. Given a set of bug fixes $\{(X_i, Y_i)\}_{i=1}^{n}$ each containing a buggy program Y_i and its fixed variant X_i, the inverse Brier score is computed by:

$$1 - \frac{1}{n}\sum_{i=1}^{n}(1 - P(Y_i \mid X_i))^2,$$

where $P(Y_i \mid X_i)$ is the probability of reproducing the bug Y_i given X_i. The inverse Brier score is a strictly proper scoring rule for probability distributions [GR07] with a natural interpretation in our context: A mutation operator is scored between 0 and 1, where 0 indicates that the operator never reproduces a real bug while 1 indicates that the mutation operator perfectly replicates all bugs. When employing the score with our mutation framework, we compute the reproduction probability based on the sampling distribution (i.e. the likelihood of resampling the original bug token).

Training Neural Bug Detectors. In **RQ2** and **RQ3**, we evaluate the impact of the mutation operator on the training of neural bug detectors. We hypothesize that training a neural bug detector on more realistic mutants also increases their ability to detect and repair real bugs. To test this hypothesis, we construct several experiments that consider the impact of different mutation operators on the training and evaluation of neural bug detectors for various bug types. In **RQ2**, we focus on neural bug detectors for Java bugs and in **RQ3** we extend our evaluation setup to neural bug detectors for Python and JavaScript. In our experiments, we replicate original training setup of the

neural bug detectors as close as possible, while replacing the mutation operator used for generating mutants. All neural bug detectors are trained on mutants while being evaluated on real bugs. To evaluate the ability of the neural bug detectors to *detect* and *repair* real bugs, we measure the *localization accuracy* (the percentage of buggy tokens correctly identified), the *repair accuracy* (the percentage of buggy tokens successfully repaired) together with the *joint localization and repair accuracy* (the percentage of buggy tokens correctly localized and repaired). For bug detectors that only classify individual tokens without *repairing* them, we instead employ *precision* and *recall*.

3.4.2 Mutation Operator Types

In our experiments, we consider three mutation operator types for generating binary operator replacement bugs, variable misuse bugs and function misuse bugs. In the following, we define the considered operator types in terms of a syntactic role R and a set of replacement targets C_T. Mutants are generated by replacing a token that is assigned to role R with a token defined in C_T.

Binary operator replacements (BOR) include bugs related to binary operators. To seed a BOR bug, we replace a random binary operator ($R = \texttt{Binary}$) with another operator from the set of binary operators:

$$C_T = \{\texttt{==}, \texttt{!=}, \texttt{<=}, \texttt{>}, \texttt{+}, \texttt{-}, \dots\}$$

Variable Misuse bugs (VarMisuse) occur when a developer mistakenly uses one variable while meaning another variable defined in scope [ABK17]. To seed a VarMisuse bug, we replace a random variable *usage* ($R = \texttt{Var\_Use}$) with another defined variable:

$$C_T = \{t_j \mid role(t_j) = \texttt{Var\_Def}\}$$

Function misuse bugs (FunctionMisuse) occur when a developer accidentally calls the wrong function, which does not fit its use case. To introduce a function misuse bug, the mutator selects a function call ($R = \texttt{Func\_Call}$), while replacing it with another function name:

$$C_T = F_{1000} \cup \{t_j \mid role(t_j) = \texttt{Func\_Call}\}$$

Following Patra and Pradel [PP21], the mutation operator can select locally occurring functions or one of the 1000 most common function names (F_{1000}).

Valid Mutations. Finally, to derive the set of valid mutations for each operator type, we define the set of valid mutations for a given program T as follows:

$$M_T = \{\langle l, r\rangle \mid role(t_l) = R, r \in C_T, t_l \neq r\}$$

3.4.3 Mutation Operator Baselines

To study the effect of mutation realism on the training of neural bug detectors, we employ several mutation operator baselines.

Loose mutation operators represent the current best practice for training neural bug detectors [PS18, VKM+19, HSS+19]. To mutate a token at a given location, a loose mutator uniformly samples a replacement independent of the mutation context. The sampling process is unrestricted, i.e. each valid mutation can be sampled with equal probability. This would for example allow mutations that replace a conditional operator `==` with an arithmetic operator `+`.

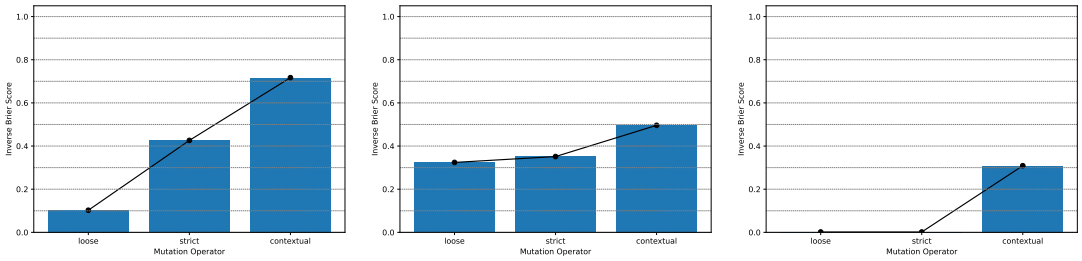
Strict mutation operators are more inline with mutation operators found in mutation testing. For example, a strict mutation operator for binary operator bugs might restrict mutations of relational operators to replacements with other relational operators. In our experiments, we seed binary operator bugs by arithmetic (AOR), relational (ROR), conditional (COR) and bitwise (BIT) operator replacements [Jus14]. Identifier bugs are less common in mutation testing. Therefore, we construct a strict mutation operator baseline for VarMisuse and FunctionMisuse by defining the following sampling distribution:

$$P(t \mid t_m, T) = \frac{exp(c(t_m, t))}{\sum_{v \in C_T} exp(c(t_m, v))},$$

where $c(t_m, t)$ is a cosine similarity between learned word vector representations for t_m and t. For learning word vectors, we employ `fasttext-cbow` since the cosine similarity between these word vectors have been shown to correlate the best with human sense of identifier similarity [WRP21].

Contextual mutation operators are generated context-dependently through our proposed framework. We employ CodeBert [FGT+20] as our masked language model. CodeBert is a pre-trained 125M parameter Transformer-based masked language model trained on six programming languages including Java, Python and JavaScript. We also experimented with unidirectional language models [LGR+21] and expert models specifically trained for a mutation task, but found that CodeBert is generally better suited for finding contextual replacements. Contextual mutators are conditioned on the code context surrounding the mutation location (limited to maximally 512 tokens by the employed language model).

Generating mutants. For training the neural bug detectors, we populate code corpora of likely correct code with artificial bugs produced by our evaluated mutators. For example, for generating mutants in Java, we consider the Java portion of the CodeSearchNet corpus (as shown in Table 3.2) consisting of more than 500K functions split into train, test and validation sets. To avoid future train-test duplicates with our real world benchmark, we employ the deduplication method by Allamanis et al. [All19] and remove all examples that appear in the test benchmark. The code corpora employed for



(a) Binary operator replacement (b) Variable misuse (c) Function misuse

Figure 3.5: Effect of the mutator on the reproducibility of real world bugs.

Python and JavaScript can be found in Table 3.2. Based on this cleaned code corpus, we construct multiple variants of the dataset by varying the injected bug type and the employed mutator. Following previous work [HSS+19], we only consider functions with at least two locations for mutations and mutate each function maximally three times at different locations. To train the mutator to distinguish correct examples from mutants, we pair each mutant with its unmutated counterpart. This results in a balanced dataset of likely correct code and mutants.

3.5 Results

In the following, we present our experimental results to answer our research questions.

3.5.1 RQ1 - Are contextual mutants more realistic?

To evaluate whether mutants generated by our contextual mutation operator are more realistic, we test the evaluated mutation operators on our reproduction task using our real world benchmark.

Experimental setup. To measure how likely a mutation operator reproduces real bugs, we employ the real bug fixes from our benchmark. As a starting point for the mutation process, we employ the fixed variant of the program *after* the bug fix. Then, we measure the probability of reproducing the original bug *before* the bug fix given the bug location. We report the inverse Brier score for reproducing the original bug. To answer our research question, we assume that mutation operators that are more likely to reproduce real bugs also generate more realistic mutants.

Results. Our experimental results are depicted in Fig. 3.5. The compared mutation operators are listed on the X-axis, while the blue bars represent the inverse Brier score. If we compare the inverse Brier score for the different mutation operators, we can make the following observations.

Contextual mutation operators are more likely to reproduce real bugs. For all evaluated bug types and operator types, the contextual mutation operator is most likely to repro-

duce the real bug given the bug location. The magnitude of improvement depends on the tested bug type and ranges between 1.5 for variable misuses to over 191 for function misuses in comparison with the loose mutation operator baseline. In addition, we find that our experiments confirm our intuition: The loose, strict and contextual mutation operators are increasingly more likely to reproduce real bugs (with the exception of function misuse, where the strict mutator achieves comparable results).

Context is important for reproducing identifier bugs. While context is important for all evaluated bug types, we observe a large performance gap between strict and contextual mutation operators for identifier-related bugs. Motivated by this observation, we manually investigated the mutation that are most likely generated for our identifier-related benchmark tasks. A typical example can be found in the second row of Table 3.3. The identifier names of bug and bug fix are unrelated, which makes it difficult for our strict mutator. The contextual mutator, in contrast, ranks `s` high as a replacement for `disables` likely because both types `String` and `List` support the contains operation.

Although the contextual mutation operator is most effective in reproducing real bugs, there exists examples where the contextual mutation operator fails. Some examples for which the contextual mutation operator fails are shown in rows 4 to 6 in Table 3.3. For example in row 5, both `file` and `alluxiUri` are both valid replacements to generate a variable misuse bug. The contextual mutation operator however prefers to replace `alluxiUriToLoad` with `file` as it is more common in this context (i.e. it is more common to use a file object with `getPath`). The strict mutation operator is more likely to reproduce the original bug as the identifiers `alluxiUriToLoad` and `alluxiUri` are more closely related.

It is important to note that the shown failure cases all represent realistic bugs that could have appeared in a real code base. In other words, our analysis shows that there exists a variety of possibilities to generate realistic bugs. In addition, our quantitative analysis has shown that our contextual mutation operator is more likely in reproducing the real bug in most cases. This also indicates that a contextual mutation operator is more likely to generate realistic mutants.

Based on these observations, we arrive at the following conclusion:

> Contextual mutation operators are on average more likely to reproduce real bugs. Thus, we can expect that mutants generated by contextual mutation operators are on average more realistic than mutants generated by non-contextual mutation operators.

3.5.2 RQ2 - Impact on the training of Neural Bug Detectors

Since we have established in RQ1 that contextual mutants are on average more realistic than non-contextual mutants, we can now evaluate whether training on more realistic mutants leads to bug detectors that are more effective on real bugs.

Table 3.3: Examples for real world bugs compared with bugs produced by our contextual mutation operator. The first three are examples for a successful reproduction of the real bug. This is followed by three cases where the mutation operator fails (real bug in comments).

Correct code	Mutant
`if(contentLength >= 0)`	`//Real bug: >` `if(contentLength > 0)`

`String s = ...`
`List<String> disables = ...`
`...`
`if(!disables`
`  .contains(a.getName( )))`
	`String s = ...`
	`List<String> disables = ...`
	`...`
	`//Real bug: s`
	`if(!s`
	`  .contains(a.getName( )))`

`results = connector.apply(`
`  context`
`);`
	`//Real bug: execute`
	`results = connector.execute(`
	`  context`
	`);`

`if(media.getDuration()<=0)`	`//Real bug: ==` `if(media.getDuration()>0)`

`alluxioUriToLoad =`
` alluxioUri.join(file);`
`...`
`new AlluxioURI(`
` alluxioUriToLoad.getPath()`
` );`
	`alluxioUriToLoad =`
	` alluxioUri.join(file);`
	`...`
	`//Real bug: alluxioUri`
	`new AlluxioURI(`
	` file.getPath()`
	` );`

`if( ... &&`
`  bulkInsertableMap`
`  .containsKey(...)`
	`//Real bug: get`
	`if( ... &&`
	`  bulkInsertableMap`
	`  .contains(...)`

Table 3.4: Results on the Java real world benchmark for detection and repair (Best results marked in bold).

Mutator	Java								
	BOR			VarMisuse			FunctionMisuse		
	Joint	Loc	Repair	Joint	Loc	Repair	Joint	Loc	Repair
Loose	9.00	14.67	44.90	28.51	29.74	69.62	1.09	2.79	21.53
Strict	15.37	**27.01**	45.75	28.99	31.68	68.20	1.50	2.52	22.70
Contextual	**16.97**	25.89	**48.28**	**33.85**	**36.67**	**72.90**	**5.49**	**8.68**	**26.08**

Experimental setup. To measure the impact of the mutation operator, we require a neural bug detector that supports the detection and repair of all bug types considered in this work. We therefore employ the joint architecture proposed by Vasic et al. [VKM+19]. During our experiments, we follow the extensions and training setup proposed by Hellendoorn et al. [HSS+19] as close as possible. In particular, we report results for a six-layer Transformer architecture [VSP+17]. As this neural bug detector was originally designed for the detection and repair of variable misuse bugs, we adapt the architecture to support single token bugs in general. More precisely, we extend the model (similar to CopyNet [GLLL16]) to include an external vocabulary into its prediction (which is helpful for repairing binary operator bugs). More details on the final neural bug detector architecture can be found in Section 2.3.

Results. Our results on the real world benchmarks for bugs found in Java projects can be found in Table 3.4. We report the key metrics in percent for each approach and highlight the best results in bold. Overall, we make the following observations.

Training on contextual mutants outperforms alternative mutation strategies. Compared to the training on strict mutants, the performance when trained on contextual mutants increases by up to 6% on all metrics and bug types, except for the localization accuracy of BOR bugs. For BOR bugs, we observe a slight tradeoff between the localization performance and repair performance (which increases by 2.5%). Still, the neural bug detector trained on contextual mutants is more effective in the detection *and* repair of real BOR bugs. In other words, by training on contextual mutants it is now possible to detect and repair more than 30 new BOR bugs that we were previously deemed impossible.

Reproduction performance correlates with neural bug detector's effectiveness. If we compare the performance of the neural bug detectors trained with different mutation operators and the reproduction performance of these mutation operators, we find that the reproduction performance correlates with the effectiveness of a neural bug detector to detect and repair real bugs. On average, switching from a loose mutation operator to a strict mutation operator and from a strict mutation operator to a contextual mutation operator improves the performance each time (up to 6% in joint localization accuracy between loose and strict mutation operator and again up to 5% further improvement

Table 3.5: Results on the real world benchmark for detection and repair (Best results marked in bold).

Mutator	Python			JS	
	VarMisuse			BOP	
	Joint	Loc	Repair	Prec	Rec
Loose	12.58	**19.40**	24.96	45.45	7.95
Strict	-	-	-	-	-
Contextual	**14.69**	19.02	**26.24**	**54.31**	**38.46**

when using contextual mutation operators).

Overall, we find that training on more realistic mutants (i.e. with a mutation operator that is more effective in reproducing real bugs) improves the bug localization and repair performance. Interestingly, since our contextual mutation operator produces the most realistic mutants (according to our metrics), the neural bug detectors trained on contextual mutants also achieve the highest performance. As a consequence, we conclude:

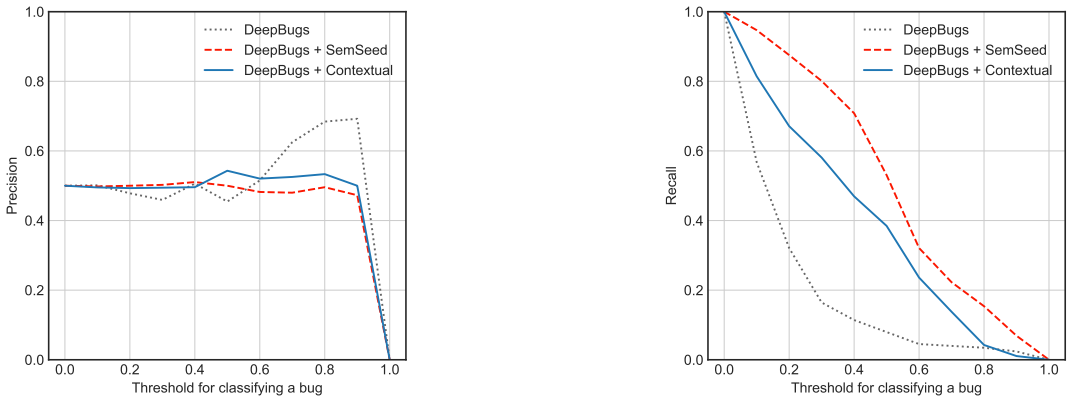
Training on more realistic mutants can significantly improve the real bug detection and repair performance of neural bug detectors.

3.5.3 RQ3 - Transfer to other languages and bug types

In RQ1 and RQ2, we mainly focused on Java bugs. To evaluate whether our results also generalize to other bug types and programming languages, we explore two additional neural bug detection approaches for Python and JavaScript. For JavaScript, we additionally compare with SemSeed which is a bug seeding technique specifically trained to invert bug fixes [PP21].

Experimental setup. For Python, we employ the same type of neural bug detector that we used for Java bugs. However, this time we train the neural bug detector on the original task of detecting variable misuse bugs in Python. To achieve comparable results with previous studies, we do not employ our extensions for general bug detection but also facilitate the more specialized two-pointer architecture used by Hellendoorn et al. [HSS+19]. For JavaScript bugs, we employ DeepBugs [PS18] which was trained for detecting binary operand bugs (BOp). The original version of DeepBugs was trained on randomly seeded bugs [PS18]. In addition, we also compare with a version of DeepBugs trained on semantically seeded bugs which are introduced by SemSeed [PP21]. Our contextual mutation operator employs the same set of mutation candidates as proposed by SemSeed. Finally, since DeepBugs classifies individual operands, we report *precision* and *recall* for detecting individual buggy operands.

Results. Our main results for Python and JavaScript bugs are reported in Table 3.5.



(a) Precision

(b) Recall

Figure 3.6: Precision and Recall for DeepBugs trained on random bugs (grey dotted), bugs introduced by SemSeed (red dashed) or contextual mutants (blue).

We again report key metrics in percent, while highlighting the best results in bold. For JavaScript, we additionally compare the precision and recall curves of different versions of DeepBugs in Fig. 3.6 at varying decision thresholds used to identify a buggy operand. Overall, we observe the following.

Training on more realistic mutants can improve the performance on real Python and JavaScript bugs. The training on contextual mutants improves all metrics, except the localization of VarMisuse bugs (with a decrease of 0.38%). For JavaScript, using contextual mutants clearly improved the precision and recall. Using the default threshold of 0.5, the number of bug detected increases from 7.83% to 38.46%.

Contextual mutants that explicitly imitate real bugs can further improve the bug detection performance. Under the precondition that sufficiently many bugs for training SemSeed are available, we find that DeepBugs trained on bugs seeded by SemSeed further improves the recall rate (up to 15% at a threshold of 0.5). A key assumption of our work is however that bug fix collections of a sufficient size do not exist for every bug type. In this case, where bug fix datasets of a sufficient size are not available, our contextual mutation operator can significantly improve the detection of real bugs without access to real bugs.

To summarize our observations, we find that contextual mutations can improve the performance in various languages, without requiring any additional real bugs as training examples. However, as seen by SemSeed, integrating real bugs into our contextual mutation framework might further improve the generation of mutants. We leave this open for future research.

We observe that the impact of training on more realistic mutants on the bug detection performance on real bugs can transfer between bug types and programming languages.

3.6 Threats to Validity

In this section, we discuss potential threats to the validity of our experiments. In our discussion, we differentiate between threats to external, internal and construct validity.

External validity. Our experiments focused on neural bug detectors for single token bugs in Java, Python and JavaScript. Therefore, our experimental results might not generalize to other bug types, programming languages or even other types of neural bug detectors.

Applicability to Other Bug Types. Since our evaluation is limited to certain bug types and available data, our results might not generalize to other bug types or other evaluation sets. To still achieve representative results, we evaluated our methodology in various different experimental setup by varying the bug type injected and by employing specialized evaluation benchmarks for each bug type. In addition, all our studied bug types are frequent for the respective language (they belong to the most frequent bug types found in the top 1000 most popular Github projects in the respective language) and are commonly explored by previous studies in neural bug detection. Still, our experimental results might not generalize to other bug types that are not considered in this work or other set of evaluation tasks.

Impact in Different Programming Languages. In our experiments, we considered mutation operators for different bug types in three different languages. Even when considering the same bug type, the mutation operator might impact the training of neural bug detectors differently in other programming languages (e.g. see VarMisuse in Java in Table 3.4 and in Python in Table 3.5). Although our experiments show that that our experimental results are transferable between languages, there may be programming languages for which our results are not transferable.

Design of Neural Bug Detector. The design of the neural bug detector might have significantly influenced our experimental results. While we considered three different types of neural bug detectors that are commonly used for detecting bugs in Java, JavaScript and Python, there might exist (future) neural bug detection architectures for which our results do not generalize.

Internal validity. The goal of experiments is to isolate the effect of mutation operators on the training of neural bug detectors. The following characteristics of our experiment design might still have influenced our experimental outcome.

Randomness of the Mutation Process. While mutation is an effective method for generating training examples at a large scale, the process of mutation is still inherently random. Therefore, applying the same mutator on the same code base might result in a different training dataset. To mitigate this problem and its impact on the validity of our evaluation, we specifically designed our experiments to be as deterministic as possible. In the evaluation of RQ1, we directly evaluate the sampling distribution of each

mutator without requiring a random sample. In experiments where a random sample is necessary, e.g. for dataset generation in RQ2 and RQ3, we fix the random seed for mutation and publish all random samples (training datasets) for replication. Even with a fixed seed, the initial process of generating the training datasets requires some form of simulated randomness. To account for statistical variance during this process, we generate large scale datasets with up to 2M examples employed for training neural bug detectors.

Randomness of the Training Process. Training neural network based bug detectors often requires the application of some variant of stochastic gradient descent. Therefore, the training process is partially random and training the same neural bug detector twice (with the same hyperparameters on the same dataset) might result in two different neural bug detectors with different evaluation results. To mitigate this issue, we also fix the seed during the training of the neural bug detectors. We publish all trained neural bug detectors to ensure that our experimental results can be replicated.

Implementation bias. The precise details of our implementation choices could have influenced our experimental results. To mitigate the impact of our implementation choices on the experimental outcome, we choose to implement all mutation operators and all neural bug detectors used in RQ1 and RQ2 in a joint framework. For RQ3, we partially used pre-existing neural bug detector implementations. To isolate the effect of the mutation operator, we only compared neural bug detectors that used the same implementation trained on same bug type. However, the implementation itself is still a research prototype. Therefore, undiscovered implementation mistakes might have influenced our experimental results.

Construct validity. A key assumption of our work is that the ability of reproducing real bugs correlates with the ability of generating realistic mutants. To ensure that this assumption holds, we performed extensive manual inspections of the mutants generated by the individual mutation operators. In addition, our experiments indicate that mutation operators that are more effective in reproducing real bugs also produce better training examples for the training of neural bug detectors. Although there exist several indicators that our assumption is true, the ability of reproducing bugs might still not directly correlate with the ability of producing realistic mutants in different setups.

3.7 Related Work

In this chapter, we introduced a novel contextual mutation operator and evaluated the impact of the mutation operator on the training of neural bug detectors. In the following, we discuss closely related works that also address the design of more effective mutation operator and that target the training of neural bug detection and automatic repair models.

Artificial bugs in the training of neural bug detectors. Artificial bugs are common in the training of neural bug detectors [PS18, ABK17, VKM+19, HSS+19, PP21, RW22a, AHO24]. DeepBugs [PS18], for example, have been trained to detect bugs in binary operator expression or function calls in popular JavaScript projects by training on artificially generated examples. Allamanis et al. [ABK17] introduced the task of variable misuse detection and trained graph neural networks on artificially modified programs to identify variable misuses. Vasic et al. [VKM+19] addressed variable misuse bugs in Python functions by introducing a joint architecture for detection *and* repair. Hellendoorn et al. [HSS+19] explored different neural architectures for the detection and repair of variable misuse bugs. While it seems that the main advances in neural bug detection are achieved by exploring new neural bug detection models, we found that the training data used during training has as well a significant effect on the performance of neural bug detectors. By training on more realistic mutants, we achieved significant performance gains for all evaluated neural bug detectors that are either based on DeepBugs or based on the models explored by Hellendoorn et al.

Even though most existing collection of real bugs are too small to be useful for training, approaches that induce bug patterns from real bug fixes [PP21, AHO24] can also be a promising strategy for generating realistic mutants. In fact, our experiments have shown that techniques that imitate real bugs can further improve the ability of a neural bug detectors to detect real bugs. These techniques however require the availability of sufficiently many real bugs for training, which is in practice often difficult to achieve (especially for more infrequent bug types) [KS20]. Our contextual mutation operator can be used as a direct alternative for existing mutation operators without requiring real bug fixes for training. Since obtaining real bug fixes at a sufficient scale for all possible bug types is a significant problem, concurrent work [AJFB21] have also explored unsupervised methods for generating realistic training examples. For example, Allamanis et al. [AJFB21] showed that training a mutator that generates hard to detect bugs can improve the performance of a neural bug detector. They however found that the introduced mutations are indeed hard to find, but often less realistic. Kanade et al. [KMBS20] and Bui et al. [BWH22b] tried to pre-train on coding tasks that are not directly related to bug detection to improve the performance of neural bug detectors. Dinh et al. [DZT+23] experimented with using large language models directly (without training) as neural bug detectors for detecting bugs in partial incomplete code.

Artificial bugs in the training of automatic program repair techniques. While neural network based approaches for automatic program repair (APR) are traditionally trained on real bug fixes [CKT+21, JLT21a], recent work has explored the effect of training with artificial bugs [YML+22, SFYM23, YM24]. For example, Ye et al. [YML+22] trained an APR system solely on randomly generated mutants which they found also to be useful for training iterative repair strategies [YM24]. Silva et al. [SFYM23] used back-translation to imitate real bug fixes. In contrast, our contextual mutation operator uses

masked language model to generate training examples for neural bug detectors. Since our experiments showed that training on contextual mutations also boost the repair performance of neural bug detectors, it could be interesting to apply our technique for APR systems in future work.

Context-dependent mutation operator in mutation testing. Although most existing mutation frameworks use mutation operators based on pre-defined mutation rules [Jus14, ABJS16], the use of contextual information in the mutation process has been explored in mutation testing [JKA17]. Just et al. [JKA17] inferred the mutant utility based on parent and child AST nodes. Allamanis et al. [ABJS16] used an n-gram language model to select mutants that appear *unlikely* in the context. Instead of exploiting the AST structure, our method can directly exploit natural hints in the code to produce contextual-dependent mutants. In addition, our contextual mutation operator uses a masked language model to select *likely* replacements in the given context. Our experiments indicate that this can effectively lead to more realistic mutants.

Generating realistic mutants can also be useful for mutation testing. In fact, subsequent work [KDPT23, DP22, GDPT24] found that generating mutants with the help of a masked language model can significantly boost the effectiveness of mutation testing frameworks. In other words, they showed that a test suite that is more effective in detecting contextual mutants is often more effective in detecting real bugs. This result suggests that our evaluation result might generalize across disciplines, and it would be interesting to evaluate our contextual mutation operator in other disciplines in future work.

3.8 Conclusion

This chapter explored the impact of training on more realistic mutants on the performance of neural bug detectors. For generating more realistic mutants, we proposed a novel contextual mutation operator that uses masked language model to inject more realistic contextual mutations. Our evaluation on thousands of real world bugs showed that our contextual mutation operator is indeed more effective in reproducing real bugs than non-contextual variants. As a result, we found that training on mutants generated by our contextual mutation operator can significantly improve the performance of existing neural bug detector in the detection and repair of real bugs.

Beyond the training of neural bug detectors, we believe that contextual mutation operators can also complement existing mutation operators in mutation testing (which was partially confirmed in subsequent works [DP22]). Our contextual mutation operator cannot only provide novel but also more fine-tuned mutations. We further see potential in the mutation of identifiers, which – with notable exceptions [MDF+01] – have so far only been sparsely investigated in mutation testing.

Limitation. Although we achieved promising results by training on contextual mu-

tants, there exists several limitations of our contextual mutation operator that could have limited its ability to generate realistic bugs. For example, our contextual mutation operator selects mutation locations randomly. Although we have shown that this can lead to realistic mutants, recent works [JLZ+22] suggest that there exists mutation contexts where real bugs appear more frequently. Therefore, by sampling the wrong mutation location, the resulting mutant might be determined to be non-realistic even before we select a mutant. While we could have built upon previous work and learn *likely* mutation locations from real bug fixes, we would still be limited by the size of existing bug fix collections. Instead, in the following chapters, we focus on *mining* massively large collections of real bug fixes. These collections could then be used for training mutation operators, but are also large enough to train neural bug detectors directly.

4 Mining Realistic Bugs

Having explored the impact of more realistic training examples on the performance of neural bug detectors in the previous chapter, we now turn to the underlying problem which we believe is the main reason for the limited performance of neural bug detectors: *the lack of realistic bugs at a sufficient scale*. While there exist several collections of realistic bugs mined from public repositories, they are often limited in size, making them difficult to use for the training of neural bug detectors. In this chapter, we investigate an alternative mining approach that allows us to mine realistic bugs from more than 500K Python Git projects. As a result, we obtain collections of real bugs in the order of millions of examples.

4.1 Motivation

Software bugs come in many forms. Some can easily be fixed by modifying a single line or statement. Others are so complex that they require complete rewrites. Especially those *smaller* bugs that appear in a single line or statement can be easily overlooked by a developer. Therefore, to relieve the developer from the burden of manually finding and fixing these simple, small bugs, many approaches have been proposed that target the automatic detection [ABK18, PS18, HSS+19, PC22, RW22b] and repair [CKT+21, LPP+20a, VKM+19] of these bug types.

However, a key problem in the development of these methods is the need for large collections of known bugs. Existing collections such as Defects4J [JJE14] or BugsInPy [WSL+20] only contain a few hundred examples. The subset of bugs that relate to specific bug types is often significantly smaller. Therefore, techniques that are more *specialized* to particular types of bugs often require custom datasets for evaluation [HP18] or have to rely on *artificial* benchmarks [DHK+16].

With the rise of data-driven methods [ABK18, PS18, HSS+19, VKM+19, LPP+20a, CKT+21, RW22b] for automatic bug detection and repair, the problem of obtaining a

sufficient number of real bugs has become even more dire. These methods often require huge datasets with millions of known bugs for training. Collecting single statement bugs at the required scale is however difficult as they rarely occur in open source projects. Therefore, existing methods (including ours in Chapter 3) have to rely on *artificial bugs* for training.

In this chapter, we address the lack of available training examples for Python by creating two ultra large collections of single statement bug fixes. For this, we mined over 500K Python Git repositories for bug-fixing code changes that modify only a single statement. By following a mining process close to previous work [KPBH21], we obtained a dataset of over 2.3M single statement bug fixes which we call `CSSB-2.3M`[1]. Since not all collected patches fix a bug in isolation, i.e. the bug fix might be *entangled* [HZ13] with unrelated code changes, we also explored a more restrictive definition of single statement bug fixes. More precisely, we filtered our datasets for bugs that can be fully patched by modifying a single statement. This process lead to our second collection of nearly 0.9M "true" single statement bug fixes called `CTSSB-0.9M`.

Relation to TSSB-3M [RW22c]. Both `CTSSB-0.9M` and `CSSB-2.3M` are derived using a mining method that is very similar to the method used to create TSSB-3M and SSB-9M. We however noticed after publishing TSSB-3M and SSB-9M that the datasets still contained a significant amount of *duplicate commits*. To obtain `CTSSB-0.9M` and `CSSB-2.3M`, we therefore employed a more aggressive deduplication scheme on top of TSSB-3M and SSB-9M to obtain their cleaned variants. We discuss our improved deduplication process in Section 4.3.1.

4.2 Single Statement Bug Fixes in the Wild

Before we describe our mining process in Section 4.3, we want to provide some characterization of the bug fixes we are interested in. In the process, we also introduce some techniques that we employed in our mining process.

Single Statement Bug Fixes. Our goal in this section is to crawl the commit history of Python projects for single statement bug fixes. For this, we view the commit history of a Python project as sequence of commits $C_1, \ldots, C_n$. For simplicity, we assume that each commit C_i modifies a single program P_i[2]:

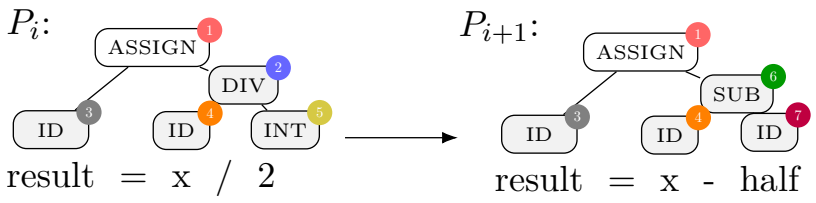
$$C_i = \quad P_i \xrightarrow{D_i} P_{i+1} \quad (M_i),$$

where P_i represents the program before the commit, P_{i+1} the program after the commit and M_i represents the commit *message* indicating the commit *intent* of C_i, e.g. to fix a bug. We use D_i to denote the difference between P_i and P_{i+1}.

[1] `CSSB-2.3M` and `CTSSB-0.9M` are "cleaned" variants of our previously published datasets SBB-9M and TSSB-3M in [RW22c].

[2] In practice, we can handle commits that modify multiple programs (e.g consisting of multiple files) as multiple commits that modify a single program.

Single Statement Changes. Since our goal are single statement bug fixes, we are mostly interested in *single statement changes*, i.e. code changes that modify a single statement. To identify single statement changes, we represent both the program before the change P_i and the program after the change P_{i+1} with their abstract syntax trees (ASTs). For example, we view the following single statement change as a modification of ASTs:



Here, we modify a single assignment statement 1 by replacing the division operation 2 with a subtraction 6:

$$D_i = \text{replace}(2, 6),$$

where replace(2, 6) replaces the complete subtree 2 rooted at the division with the subtree 6 rooted at the subtraction. We define that an AST modification D_i is a single statement change iff the replaced subtrees (2 and 6) have each at least one ancestor or are itself a node that represents a single statement (1) and there does not exist another node in the replaced subtrees that represents a statement.

Bug-Fixing Changes. It is non-trivial to decide whether a code change represents a bug-fixing change. To still decide whether a code change is *potentially* bug-fixing, we rely on a keyword-based heuristic. Similar to previous work [KS20], we define a set of keywords K_{bug} which occurrence in a commit message M_i indicate a bug-fixing change:

$$K_{bug} = \{ \text{“error”,“bug”, “fix”, “issue”, “mistake”, “incorrect”, “fault”, “defect”, “flaw”, “type”}\}$$

We assume that a commit contains a bug-fixing change if at least one keyword $k \in K_{bug}$ is contained in the commit message M_i. Note that this heuristic to determine the bug-fixing intent of a code change has been used in previous works [TWB+19c, KS20, KPBH21, CKT+21] and has been shown to be highly precise (with a precision of at least 90%). Finally, we define a single statement bug fix as a code change that is both a single statement change and a bug-fixing change.

Tangled Commits. A key problem when mining software repositories for bug fixes is the existence of *tangled commits* [HZ13]. A tangled commit is a (bug fixing) code change that modifies more code than necessary to fix a bug. Tangled commits are often difficult to detect as irrelevant code changes are often not indicated by the developers. One way to distinguish bug fixing changes from irrelevant code changes is to use a

test oracle [HZ13]. However, this is often not available in our setup. Therefore, we try to identify bug fixing commits that isolate the bug fixing code. For this, we track the changes made by a commit and we focus on commits that *only* change a single statement (without further modifications).

AST Differencing. Throughout this chapter, we differentiate bug fixes into different bug fix patterns. In the process, we mainly follow the SStuB bug patterns proposed by Karampatsis and Sutton [KS20] and later extended to Python by Kamienski et al. [KPBH21] (see Section 2.1.2). For this, we annotate each bug-fixing code change D_i with a unique SStuB pattern (if possible). During this process, we will notice that a majority of bug fixes cannot be assigned to a SStuB pattern. To have a more generic way to describe bug fixes, we also characterize code changes via *AST differencing* [FMB+14]. The key idea of AST differencing is to describe the change to the AST via so-called AST edit scripts. The AST edit script contains a list of the following four types of operations:

- `INSERT`: inserts a new AST node at a given location,
- `MOVE`: moves an existing AST node to a different location,
- `UPDATE`: updates the value of an AST node,
- `DELETE`: deletes a single AST node.

The edit operations are sequentially applied to transform the AST before the fix to the AST after the fix. A key advantage of AST differencing is that resulting edit script can accurately describe the syntactic changes needed for fixing a bug. In our analysis, we compare different bug fixes based on the AST edit operation needed to fix a given bug.

Example. If we consider again the single statement change from the beginning of this section, we can represent the code change D_i by the following AST edit script:

`INSERT` (6, 1), `INSERT` (7, 6), `MOVE` (4, 6), `DELETE` (5), `DELETE` (2)

Here, we start by inserting the new subtraction operation 6 without its children as a child of 1. Then, we build the children of 6 by inserting a new identifier 7 and moving the old identifier 4 to 6. Finally, we delete the old subtree by removing 5 and 2.

4.3 Mining Real Bug Fixes at Massive Scale

In the following, we describe our process for mining single statement bug fixes in Python Git projects in more detail. An overview can be found in Fig. 4.1. Since mining single statement changes that relate to bug fixes can become easily computationally infeasible for a large number of projects, we start by mining the commit history of Git projects for single line edits. Single line edits can be easily identified without requiring expensive AST based source code analyses. Afterwards, we filter and deduplicate the resulting

Figure 4.1: Overview over our mining process

dataset for obtaining a set of unique single statement changes. To obtain a dataset of real bug fixes, we further filter the dataset for commits where the commit message indicates a bug fixing change. To be able to navigate our datasets more easily, we annotate each bug fix with a distinct bug pattern. In Section 4.3.1, we describe the initial mining process. The different filters that we apply to obtain a dataset of real bug fixes are described in Section 4.3.2. Finally, our annotation process is described in Section 4.3.3.

4.3.1 Mining Single Statement Changes in Python Projects

To effectively mine single statement bug fixes at a massive scale, we start with a cheap approximate process to mine single statement bug fix *candidates*. Afterwards, we apply several filters to identify single statement bug fixes among the mined bug fix candidates.

Starting point: Git Repository Index. As the starting point of our mining process, we use the Libraries.io 1.6 [Kat20] package index. Libraries.io indexes over 2M Python Git repositories related to Python packages published for PyPI[3], conda[4] and other package managers. We deliberately chose not to exclude repositories based on their popularity, as was the case in previous work [KPBH21]. Although mining popular projects might ensure a well-maintained codebase, our goal is to capture real bugs that frequently occur during the development process. Some of them might only occur in less popular, less well-maintained projects, as they would be caught before arriving in better-maintained projects. In addition, we also included fork projects as they might contain bug fixes that do not appear in the original project. To avoid commit duplication due to forks, we later remove all duplicate commits from our final datasets. Note that our datasets do not include commits from all indexed Python projects since not all projects are publicly accessible or contain a single statement bug fix.

Mining Single Line Edits. Mining single statement bug fixes at the scale of millions of repositories can easily become computationally infeasible. For this reason, we decided to crawl the commit history of Python projects for *single line edits* first. A single line edit is a code change that modifies exactly one line in the program. In contrast to single statement changes, a single line edit can be easily determined at a textual level without requiring to parse the AST of the complete file. Still, in Python, single line edits and

[3]https://pypi.org
[4]https://anaconda.com

single statement changes are closely related: A single statement or a single statement header for a compound statement often corresponds to a single logical line in Python. Even if the logical line spans over multiple physical lines, we still capture changes to multi-line statements that modify the statement at a single location. However, we miss code changes that modify a multi-line statement at multiple locations, e.g. the change spans across multiple physical lines.

To determine whether a commit contains a single line edit, we compute the *textual* difference between the code before the commit and after the commit. Similar to the Unix diff algorithm, we view textual code modifications as removing old lines and adding new lines. We compute the number of modifications by comparing the removed and added lines. To reduce the number of false positives, we tokenize each line with a Python tokenizer[5] and compare the token sequences before and after the commit. By employing a tokenizer, we automatically ignore changes to the code formatting. In addition, we configure the tokenizer to ignore changes to code comments. During the mining process, we store all commits that modify a single line together with all computed file differences. We ignore all commits that either add or remove complete files since we are only interested in single line modifications.

For the mining process, we distribute the workload on a cluster with over 1000 workers. Each worker is assigned a set of Git repositories. The worker iteratively clones a given repository and crawls the commit history for single line edits. The complete mining process took around two weeks and produced a total of over 66M single line edits from more than 500K Git repositories.

Filtering for Single Statement Changes. The resulting dataset of single line edits might still contain many false positives, e.g. multi statement changes that are located in a single line. To filter out these false positives, we now employ a more precise AST based analysis to identify single statement changes. For this, we iterate over all collected commits, while analyzing the computed file differences to identify single statement changes. During this process, we excluded all commits that either (1) do not contain a single parseable statement or (2) modify more than a single statement. For identifying single statement changes, we compute the AST difference D_i based on the computed file differences. The file difference typically contains the modified code lines together with some context code lines. Although this is not sufficient to compute the complete AST of the modified files, the context is still sufficient for the identification of single statement changes. We employ a best-effort AST parser[6] to compute the (partial) AST of the code before and after the code commit. To locate the difference, we perform a simultaneous depth-first search, similar to Kampastsis and Sutton [KS20], until we find the first node where the two ASTs differ. We exclude all commits where the computed AST node is not located inside a statement or is a root for multiple statements.

[5] We use the `tokenizer` package from the Python standard library.
[6] We use the tree-sitter library to parse partial code.

Deduplication. Our initial assumption for the deduplication process was that (1) duplicate commits in fork projects share the same commit hash[7] as the original commit and (2) that fork projects have the same name as the original projects (but with different project owners). After our initial round of deduplication, we found that both assumptions (1) and (2) do not hold in practice. For example, we found that duplicate commits with different commit hash can occur due to commit squashing[8]. We also found that it is not uncommon to change the project name of a fork project. Therefore, to remove these duplicates, we additionally employed a more aggressive deduplication scheme: We identify duplicate commits solely based on previously computed file differences that contain the added and removed line together with some surrounding context lines. For the deduplication, we then remove all duplicate commit that have the exact same file difference (independent of the commit hash and the name of the fork project).

Result of the Mining Process. After completing our mining process and filtering the collected dataset for single statement changes, we now remain with a set of nearly 9M single statement changes from over 448K projects. Note that most of the single line edits are removed due to the deduplication process. Since not every single statement change relates to a bug-fixing change, we further filter the dataset in the following for single statement bug fixes.

4.3.2 True Single Statement Bug Fixes

We consider two types of bug fixes: *single statement bug fixes* and *true single statement bug fixes*. Single statement bug fixes are single statement changes with a bug-fixing intent. To decide whether a single statement change has a bug-fixing intent, we employ the keyword based heuristic described in Section 4.2.

While this heuristic is effective in identifying commits with a bug-fixing intent, it also assumes that the commit message and the code changes are related. In practice, however, bug-fixing code commits are often entangled with code changes unrelated to the bug fix [HZ13]. As a result, the commit message might indicate a bug fixing intent, but the single statement change might not be related to the bug fix, e.g. because it is part of a commit with multiple changes. To address this problem, we additionally filter our dataset for true single statement bug fixes. True single statement bug fixes are single statement bug fixes that fully patch a bug with exactly one single statement change. For the same reason we filter for true single statement bug fixes, we also avoid commit unrolling [KS20], which would split a single multi statement bug fix into multiple partial single statement fixes.

Result of the Filtering Process. After filtering our dataset for single statement bug fixes and true single statement bug fixes, we end up with two datasets `CSSB-2.3M` and

[7] A Git commit hash is a *unique* identifier for a single commit in a Git project.

[8] Commit squashing summarizes multiple commits with a single new commit. See https://git-scm.com/docs/git-rebase

CTSSB-0.9M, which contain respectively nearly 2.3M single statement and more than 0.9M true single statement bug fixes from more than 160K Python Git repositories.

4.3.3 Characterizing Bug Fixing Edits

After collecting single statement bug fixes in our datasets CSSB-2.3M and CTSSB-0.9M, we annotate each bug fix with information that will later help us to analyze our datasets. In the process, we employ two ways to characterize a single statement bug fix: (1) a SStuB pattern and (2) an AST edit script. SStuB patterns [KS20] are used to categorize single statement bugs into frequently occurring bug types. In total, we distinguish between 20 typical SStuB patterns for Python [FMB+14]. To categorize the individual bug fixes into SStuB patterns, we assign each bug fix to a unique bug pattern. If there exists multiple patterns that would fit a given bug fix, we assign the bug fix to the most specific pattern. Bug fixes that do not fit into a SStuB pattern are assigned to a generic *single token* or *single statement* pattern (depending on whether the bug fix modifies one or multiple tokens).

To be able to analyze the difference between the code before and after the bug fix more effectively, we employ AST differencing and compute an AST edit script between the AST of the code before the fix and the code after the fix. An AST edit script [FMB+14] describes the changes that need to be applied to the AST of the buggy code to arrive at the fixed code. For computing the AST edit script, we use a reimplementation of the GumTree [FMB+14] algorithm. GumTree computes for each bug fix a sequence of AST operations that consists of insertion, deletion, move and update operations.

Result of the Annotation Process. After annotating each bug fix in CSSB-2.3M and CTSSB-0.9M with a SStuB pattern and an AST edit script, we found that around 50% to 60% of all bug fixes cannot be assigned to a SStuB pattern. In addition, we found that the AST edit scripts needed to fix a single statement bug are often quite short with an average of four to five AST operations needed to fix a bug.

4.4 Dataset Analysis

Having access to single statement bug fixes from over 100K Git projects, we are now interested to gain new insights on how programmers make simple mistakes and how they fix them. For this, we analyze the bug fixes collected in our datasets CTSSB-0.9M and CSSB-2.3M. Our analysis was guided by the following research questions:

RQ1 Does the distribution of SStuBs in our datasets correlate with the distribution of SStuBs in previously collected datasets?

RQ2 How different are single statement bugs from those identified by SStuB patterns?

Table 4.1: SStub pattern statistics for CTSSB-0.9M, CSSB-2.3M, and PySStuBs

SStuB Pattern	**CTSSB-0.9M**		**CSSB-2.3M**		**PySStuBs**	
	Count	%	Count	%	Count	%
Change Identifier Used	61K	16	158K	17	9K	12
Change Binary Operand	42K	11	81K	9	4K	6
Same Function More Args	37K	10	110K	12	10K	14
Wrong Function Name	34K	9	92K	10	9K	12
Change Numeric Literal	30K	8	87K	9	5K	7
Add Function Around Expression	28K	8	56K	6	6K	9
Change Attribute Used	26K	7	70K	8	5K	7
Add Method Call	16K	4	31K	3	3K	5
More Specific If	13K	4	24K	3	2K	3
Add Elements To Iterable	13K	4	43K	5	2.5K	3
Same Function Less Args	13K	4	42K	4	3.4K	5
Change Boolean Literal	11K	3	24K	3	1.5K	2
Add Attribute Access	9K	2	18K	2	1.5K	2
Change Binary Operator	8K	2	16K	2	<1K	1
Same Function Wrong Caller	7K	2	11K	1	1.2K	2
Change Keyword Argument Used	6K	2	15K	2	1.5K	2
Less Specific If	5K	1	9K	1	<1K	1
Change Unary Operator	4K	1	6K	<1	2K	3
Same Function Swap Args	2K	<1	19K	2	<1K	1
Change Constant Type	2K	<1	3K	<1	2K	3
NoSStuB - Single Statement	295K	-	791K	-	-	-
NoSStuB - Single Token	195K	-	580K	-	-	-
Total SStubs	0.37M	100	0.92M	100	73K	100
Total	0.86M	-	2.29M	-	-	-

During our analysis in RQ1, we mainly compare our datasets `CTSSB-0.9M` and `CSSB-2.3M` with a collection of Python SStuBs collected in previous studies [KPBH21]. RQ2 should provide a more general view on the bugs contained in our datasets.

4.4.1 RQ1 - Does the distribution of bug fixes change?

Since previous work [KPBH21] mostly focused on mining SStuBs on popular Python projects, we are interested how the bug distribution changes if we include a different population of Python projects in our mining process.

Experimental setup. We measure the frequency of SStuBs in our datasets `CTSSB-0.9M` and `CSSB-2.3M` and in PySStuBs [KPBH21], a previously explored collection of Python SStuBs. For comparing the distribution of SStuBs in the three datasets, we employ the Spearman rank correlation coefficient [Spe61]. For this, we assign each SStuB pattern i to a rank $R(X_i)$ according to its frequency X_i in the dataset. Let X_i and Y_i be the frequency of SStuB pattern in two compared datasets, then the Spearman rank

correlation coefficient r is computed for n different SStuB patterns as follows:

$$r = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)},$$

where $d_i = R(X_i) - R(Y_i)$ is the difference between the two rankings of X_i and Y_i. We say that the *ranks* of SStuBs in two datasets are correlated iff they have a rank correlation close to 1.

Results. The number of occurrences of each pattern in the three datasets are reported in Table 4.1. We additionally report the number of single statement bug fixes that cannot be classified as a SStuB (NoSStuB). Since PySStuBs only collects SStuBs, we report only the number of SStuBs for PySStuBs. In general, we find that the distribution of SStuBs changes slightly between datasets. The rankings of pattern frequencies are however highly correlated with a Spearmann rank correlation of 0.82 between `CSSB-2.3M` and PySStuBs and 0.84 between `CTSSB-0.9M` and PySStuBs. As expected, the two datasets `CSSB-2.3M` and `CTSSB-0.9M` show a significantly higher correlation with a Spearmann correlation coefficient of 0.94. Interestingly, we find that high frequency patterns such as *Change Identifier Used*, *Same Function More Args* and *Wrong Function Name* remain highly frequent across datasets. *Change Binary Operand* bugs become more dominant in the `CTSSB-0.9M` dataset. This could indicate that these types of bugs are harder to identify by a developer and, hence, are more likely to be fixed after the main development phase in an independent fix. Finally, we find that *only 40% - 43% of all Python single statement bugs* fit a SStuB pattern.

In total, we find that:

> The distribution of bugs shifts slightly between datasets. The rankings of pattern frequencies are however highly correlated such that high frequency patterns remain highly frequent across bug collections.

4.4.2 RQ2 - How different are bugs that do not classify as SStuBs?

During our analysis, we observed that most of all single statement bugs do not classify as a SStuB (NoSStuB - Single Statement and NoSStuB - Single Token in Table 4.1). With RQ2, our goal is to gain some insights in how single statement bugs that does not classify as a SStuB – which we call NoSStuBs here – relate to existing SStuB patterns.

Experimental setup. To analyze the similarity between SStuBs and NoSStuBs, we compare the edit operations needed to fix a SStuB and operations needed to fix a NoSStuB. For this, we abstract the concrete edit operation needed to fix a bug, e.g. we abstract the operation `INSERT`(6, 1) from our example in Section 4.2 by representing it as an `INSERT_SUB_IN_ASSIGN` operation. Then, we compute for each edit script E a

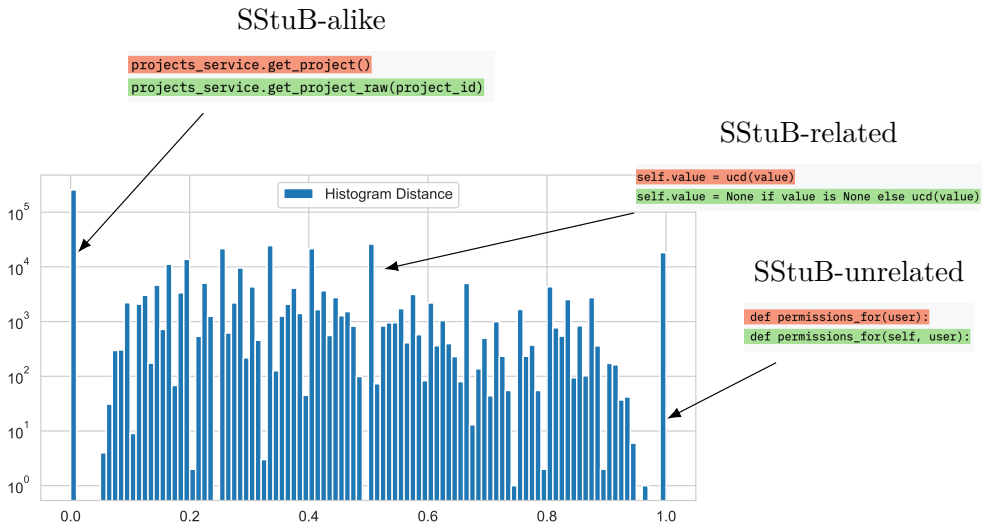


Figure 4.2: Similarity of NoSStuBs with SStuBs given as the Jaccard distance between edit operations. The x-axis corresponds to the binned distance to the most similar SStuB bug and the y-axis corresponds to observed frequencies.

set of abstract edit operations A_E, e.g. for our example:

$$A_E = \{\texttt{INSERT\_SUB\_IN\_ASSIGN}, \texttt{INSERT\_ID\_IN\_SUB}, \\ \texttt{MOVE\_ID\_TO\_SUB}, \texttt{DELETE\_ID}, \texttt{DELETE\_DIV}\}$$

To then compare the similarity of NoSStuBs and SStuBs, we employ the minimal Jaccard distance J_δ [Jac12] between the sets of abstract edit operation:

$$J_\delta(A_{E_N}) = \min_{S \in SStuBs} \frac{|A_{E_N} \cup A_{E_S}| - |A_{E_N} \cap A_{E_S}|}{|A_{E_N} \cup A_{E_S}|},$$

which represents the minimal distance of set of abstract edit operations A_{E_N} for a NoSStuB N to the closest set of abstract edit operations of a SStuB S in the dataset. Note that a distance of 0 ($J_\delta(A_{E_N}) = 0$) indicates that there exist a SStuB in the dataset that uses the exact same (abstract) edit operations as the NoSStuB. A distance of 1 indicates that the NoSStuB does not share any edit operations with a SStuB in the dataset. In the following, we focus on NoSStuBs in `CTSSB-0.9M`. The results for `CSSB-2.3M` are similar.

Results. A histogram of all Jaccard distances between NoSStuBs and SStuBs in `CTSSB-0.9M` are shown in Fig. 4.2. The distances allow us to categorize NoSStuBs into approximately three classes: SStuB-alike (distance of 0), SStuB-related (distance of around 0.5) and SStuB-unrelated (distance of 1). We find that SStuBs and SStuB-alike bugs make up around 73% of the dataset and can hence be fixed by the same type of edit operations. Another frequent class are SStuB-related bugs. SStuB-related bugs are related to existing SStuB-patterns but are not classified as SStuBs. An example

are inline if-conditions. While being fixed similar to regular if-statements, they are not covered by any actual SStuB pattern. We find that around 2% of all collected bugs are SStuB-unrelated. SStuB-unrelated bugs require edit operations that are not covered by any SStuB pattern. This are often edit operations that are highly specific to the Python language. For example, developers commonly forget to add the `self` argument to a Python method definition. The fact that this class of bugs is infrequent is however very promising for automatic repair methods, i.e. methods that support the edit operations needed to fix SStuBs are also well suited to repair a wide range of single statement bugs.

Overall, we find that:

> Most of the bugs that are not classified as SStuBs are at least SStuB-alike or SStuB-related. Only around 2% of all single statement changes require (abstract) edit operations which do not occur during fixing SStuBs.

4.5 Threats to Validity

In this section, we discuss potential threats to the validity of our mining process and the outcomes of our dataset analysis.

External validity. During our mining process, we focused on single statement bug fixes in Python Git repositories. Since our analysis is based on the real bug fixes found during the mining process, our analysis result might not generalize to other bug types, other programming languages or other types of projects than those considered during our mining process.

Other Types of Bugs. During our analysis, we mostly focused on properties that are related to existing SStuB patterns. SStuB pattern relate to frequently occurring bug types in the respective programming language and are commonly explored by previous studies in bug detection and automatic repair. Still, our analysis results might not generalize to different types of bug patterns. In particular, our results might not generalize to multi-statement bugs which might require different types of edit operations to be fixed.

Other Programming Languages and Projects. During our mining process, we restrict ourselves to real bug fixes found in public Python projects. The concrete instantiation and frequency of bugs might vary for other programming languages and projects. For example, our analysis showed that around 2% of the dataset are SStuB-unrelated and often highly specific to Python. Therefore, it is likely other programming languages encourage other types of bugs that are more specific to the respective language. Although we already mined a large portion of the publicly available Python projects, we were restricted to Python projects that are publicly accessible. The distribution of simple bugs in private projects or projects that do not have version control might be different.

Internal validity. The goal of our analysis is to gain some insights in the distribution of single statement bug fixes in Python projects. However, the following characteristics of our mining process might have influenced the outcome of our analysis process.

Identification of Real Bug Fixes. Even though the employed heuristics for identifying bug fixing commits has been repeatedly shown to be highly precise, there is still a chance for false positives in our datasets. To reduce the risk of false positives, we have designed our mining process to be as precise as possible (in our setup) by avoiding commit unrolling and by filtering for isolated bug fixes in `CTSSB-0.9M`. The latter does not only guarantee that the causal relationship between commit message indicating a bug fix and code change persists, but also avoids the mining of entangled code changes. Still, our datasets might include code changes that are flagged as real bug fixes while they do not relate to a bug fix.

Inaccuracies of the Mining Process. To scale our mining process to thousands of repositories, we used several strategies to reduce the cost of mining. Some strategies such as employing a best-effort AST parser might have introduced inaccuracies in the mining, filtering or annotation process. In addition, our implementation itself is still a research prototype. Therefore, undiscovered implementation mistakes might have influenced our analytical results.

Construct validity. A key assumption of our work is that bugs that appear frequently in the commit history of public projects are representative for bugs that appear frequently during the development process. However, to appear in the commit history of a project, a bug has to first be identified as such and a fix to the identified bug has to be committed. Therefore, our datasets might be biased towards bugs that (1) are difficult to detect during the initial development process while (2) being detectable later in the project cycle. Although we tried to address this by mining projects with varying popularity and including both well- and less well-maintained projects, the mined projects might still include bugs that are not yet discovered. In addition, there might still exist classes of bugs that appear more frequently during the development than they appear as bug fixes in the commit history of public projects.

4.6 Related Work

In this chapter, we introduced a mining process that scales to thousands of repositories which allowed us to mine two large scale bug fix datasets. In the following, we discuss related work that focuses on mining public repositories for real bug fixes.

Single Statement Bugs. Single Statement Bugs in form of SStuBs have been explored in previous work [KS20, KPBH21]. Karampatsis and Sutton [KS20] collected around 63 thousand Java single statement bugs from 1000 popular projects that fit at least one of 16 different SStuB patterns. They found that around 33% of single statement bug fixes fit at least one SStuB pattern. Kamienski et al. [KPBH21] explored similar

types of bug patterns for the 1000 most popular Python projects. During this process, they identified seven new SStuB patterns typical for Python. Their collection of bugs contains around 73K examples. Similar to Kamienski et al., we also explored bugs that fit SStuB patterns in Python. However, our mining process allowed us to collect SStuBs from more than 500K projects resulting in a SStuB dataset that is up to 13x larger. In addition and in contrast to previous bug collections, we also analyze edit operations needed to fix a bug. Therefore, we were able to explore bug types that are not covered by SStuB patterns.

Commit histories of public projects are not only a great source for mining bug fixes but can also be used for obtaining general code changes. For example, Megadiff [MMY+21], a large scale dataset around 663K code changes in Java, was mined from the commit history of 101K Git repositories. As the focus of Megadiff are general code changes, the number of potential bug fixes contained in the dataset might be significantly smaller. Recently, CommitPack [MLZ+23] was introduced which is a collection of 4 terabyte of code commits across 350 programming languages. Although the dataset is also not focused on bug fixes, due to its size it could be an interesting starting point to evaluate our filtering and annotation process for other programming languages than Python. We leave this open for future work.

Finally, subsequent works [PR23, MSC+23] has found that coding competitions such as Project CodeNet [PKJ+21] can also be a great source for mining simple bugs and their fixes.

Learning from Single Statement Bug Fixes. Collections of single statement bug fixes have already been used for training data-driven automatic repair approaches. For example, with the objective of machine-learning based program repair for single line bugs, Tufano et al. [CKT+21] collected a set of 787 thousand bug-fixing single line commits in Java. They used the same heuristics we employed for identifying bug-fixing code changes. In addition, their method is trained to translate buggy code lines into its fixed version. Bader et al. [BSPC19a] showed that AST edit scripts can be effectively employed for automatic program repair by predicting bug fixes based on previously seen AST transformations. Not only does our dataset provide almost 3x more training data (`CSSB-2.3M`), which has the potential to improve the performance of existing methods [HNP09], but also by annotating each bug fix with an AST script our dataset can directly be employed in various training setups.

That large scale datasets are a necessity for training data-driven methods has also been confirmed by subsequent works [BWH22a, PR24]. Bui et al. [BWH22a] collected a dataset of 132K and 53K bug fixes for Python and Java respectively to train a bug detection and repair system. Prenner and Robbes [PR24] used TSSB-3M (a less filtered variant of `CTSSB-0.9M`) to evaluate the impact of the surrounding context on automatic repair methods. We ourselves used `CTSSB-0.9M` and `CSSB-2.3M` as a starting point for creating a real bug fix dataset for training neural bug detectors as we further discuss

in Chapter 5.

Interestingly, recent works [HV23, HVKV24] have shown that datasets of code with known vulnerabilities [WNV+22] can be used to train large language models to generate more secure code. The authors mainly focused on common software vulnerabilities. Since large language models also tend to generate simple stupid bugs [JADM23], we believe that real bug fix datasets like `CTSSB-0.9M` and `CSSB-2.3M` could be used in the future to train large language models to generate less buggy code.

4.7 Conclusion

This chapter explored a mining process to obtain single statement bug fixes at a massive scale by mining the commits of over 500K Git repositories. As a result of our mining process, we created two new datasets called `CSSB-2.3M` and `CTSSB-0.9M`. `CSSB-2.3M` collects almost 2.3 million single statement bug fixes and `CTSSB-0.9M` focuses on nearly 900K *true* single statement bug fixes. `CTSSB-0.9M` guarantees that the included bug fixes are not entangled and that every collected bug can be fixed by modifying exactly one statement. We believe that datasets of this size could facilitate future research in data-driven bug detection and automatic repair – which was already confirmed by subsequent works [BWH22a, PR24, RW23].

Limitation. Our goal is to employ the real bug fix datasets in the training of existing neural bug detectors. However, there exist challenges when using the collected datasets for the training of existing neural bug detectors. Most existing neural bug detectors are specialized to specific types of single statement bugs such as variable misuses or binary operator bugs. As a consequence, only those real bug fixes that resemble bug types supported by neural bug detectors can be used for training. Therefore, in the next chapter, we are interested whether we can use the real bug fixes collected in `CSSB-2.3M` and `CTSSB-0.9M` for the training of existing specialized neural bug detectors.

5 Learning from Real Bug Fixes

In Chapter 3 and Chapter 4, we explored ways to obtain realistic bugs via mutation and mining. In this chapter, we are now interested how these bugs (and their fixes) can be used in the training of neural bug detectors. For this, we perform a systematic study to compare neural bug detectors trained on real bug fixes, mutants, and mixtures of mutants and real bug fixes at various dataset scales and with varying training techniques. As a result of this study, we are able to identify a training configuration that significantly improves the bug detection and repair capabilities of existing neural bug detectors.

5.1 Motivation

Real bug fixes seem to be the perfect source for learning to detect real bugs. They not only show how bugs typically occur in open source projects, but also how they are fixed by software developers. Yet, they are rarely utilized in the training of neural bug detectors. A key problem is that obtaining real bug fixes at a sufficient size for the training of neural bug detectors has been challenging in the past. Therefore, to obtain the necessary amount of data needed, neural bug detectors are often trained on *code mutants* instead of real bug fixes. These code mutants are generated by injecting small code mutations into existing code. As this process is fully automated, mutants can easily be generated at a scale that is large enough for the training of neural bug detectors. However, as we have seen in Chapter 3, mutants are not necessarily representative for *real* bugs that appear in real projects. Therefore, neural bug detectors that are trained purely on mutants usually underperform when evaluated on real bugs [AJFB21, HSS+19].

In contrast, in this chapter, our goal is to evaluate the impact of real bug fixes (at a large scale) on the training of neural bug detectors. Previous work [HBV22] has already shown that neural bug detectors significantly suffer from a distributional

shift between the distribution of artificial bugs (mutants) seen during training and the real bug distribution found in open source projects. While adapting the training distribution can significantly improve the *precision* of neural bug detectors, existing neural bug detectors remain severely limited in their ability to detect and repair real bugs. We hypothesize that – because of the scale of the employed datasets – the bug patterns needed to find and fix real bugs lack *support* in the training distribution. Therefore, it is unclear (1) how real bug fixes would impact the training process at a larger scale, (2) whether the size of the datasets (of mutants or bug fixes) employed in previous studies limited the performance of neural bug detectors and (3) whether neural bug detectors would benefit from mutants at all when a larger set of real bug fixes is available for training.

To answer these questions, we performed a systematic study comparing neural bug detectors trained on real bug fixes, mutants and mixtures of real bug fixes and mutants at various dataset scales. To evaluate the impact of real bug fixes at a large scale, we constructed a novel dataset of 33K real world Python bug fixes representing *four* common types of single token bugs typically addressed by neural bug detectors. For evaluating the impact of scale, we evaluate several neural bug detectors trained on subsets of the real bug fix dataset ranging from a few hundred examples to the full dataset size. To evaluate the impact of mutants at different dataset scales, we vary (1) the type of mutation operator used, (2) the number of mutants generated per code snippet and (3) the number of code snippets used in the mutation process. For evaluating the joint impact of training on real bug fixes and mutants, we employ a joint training framework for training the neural bug detector. For this, we train the neural bug detectors in two phases by first *pre-training* on mutants and then *fine-tuning* the pre-trained neural bug detector on real bug fixes. This allows us to not only evaluate the joint impact of mutants and real bug fixes at scale, but also individually in the same training framework. Altogether, with the help of our study, we are able to identify a training setting that significantly improved the ability of existing neural bug detectors to detect and repair real bugs.

5.2 Neural Bug Detection of Single Token Bugs

In this chapter, we investigate the impact of mutants and real bug fixes at a large scale on the performance of neural bug detectors for *single token bugs*. Therefore, before we describe our study design in Section 5.3, we provide in the following an overview of the most relevant concepts for the training of single token bug detectors.

Single Token Bug Detectors. As we have seen in Section 2.3, neural bug detectors for single token bugs are designed to (1) detect bugs in programs that can be represented as a sequence of *program tokens* $T = t_1, t_2, \ldots, t_n$, (2) localize the location of a single token t_l that likely causes the bug (if any) and (3) find a replacement r that fixes the single token bug. As in the previous chapters, we also focus here on neural bug detectors

for specific types of bugs such as variable misuse [ABK17], binary operator [PS18], unary operator [AJFB21] and literal bugs [AJFB21]. During our evaluation, we evaluate both *graph based* neural bug detectors [HSS+19] that learn from graph representations of the program and *Transformer based* neural bug detectors [VSP+17] that directly operate on the token sequence.

Mutation. To train neural bug detectors for single token bugs, previous work has mainly focused on training on *mutants*. Mutants are artificially generated by introducing an artificial bug into an otherwise bug-free program via mutation operator. As we have seen in Section 2.3.3, the mutation operator can be seen as a token replacement operation that can be inverted by the neural bug detector:

$$T \xrightarrow{\text{mutate}(t_l,r)} T_M \xrightarrow{\text{replace}(t_l,r^{-1})} T.$$

For a dataset of bug-free programs, mutants are generated by randomly replacing a token with another token of the same type. The token types (e.g. binary operators) are specified such that the mutants remain interpretable after mutation. While most existing neural bug detectors are trained on mutants that are generated purely randomly, we found in Chapter 3 that neural bug detectors can become more effective when trained on more realistic contextual mutations. Therefore, we consider also in this study the impact of using a more advanced mutation operator. See Appendix A.2 for more details on the design of the contextual mutator used in our study.

Real Bug Fixes. For training neural bug detectors on real bug fixes, we also consider *real* single token bug fixes. Here, a bug in version T_i is fixed by replacing a single token:

$$T_i \xrightarrow{\text{replace}(t_l,r)} T_{i+1}.$$

Note that real bug fixes are not necessarily complete. Therefore, while T_i contains a bug that can be fixed by replacing t_l with r, the version *after* the bug fix T_{i+1} is not necessarily bug-free.

5.3 Studying the Impact of Real Bug Fixes and Mutants at Scale

In the following, we introduce our study design. The goal of our study is to investigate the impact of (1) learning from real bug fixes and (2) learning from mutants *at different dataset scales* on the performance of neural bug detectors. In the process, we explore different *training strategies* that allow us to measure the impact of training on real bug fixes, mutants and mixtures of real bug fixes and mutants. In Section 5.3.1, we discuss the training strategies employed in our study. We discuss ways to scale up the training datasets in Section 5.3.2. The implementation of our training strategies and neural bug detectors used in our work are discussed in Section 5.3.3.

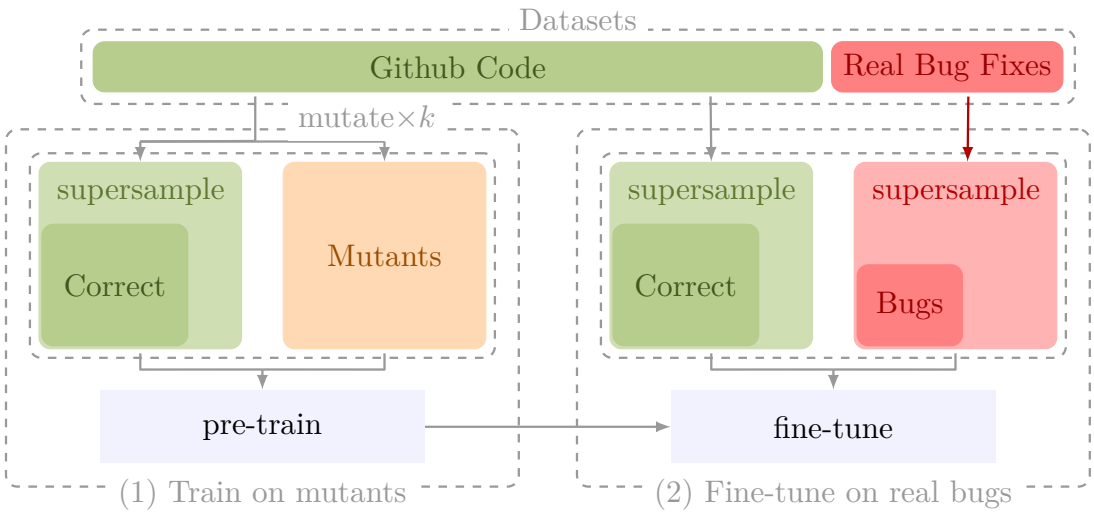


Figure 5.1: Overview of the training process

5.3.1 Training with Code Mutants and Real Bug Fixes

To isolate the impact of training with mutants and real bug fixes, we perform our training process in two phases: (1) a *pre-training* and (2) a *fine-tuning* phase. During pre-training, we train the neural bug detectors on artificially generated code mutants. The code mutants are generated by mutating existing code mined from public repositories. Then, in the fine-tuning phase, we either continue the training of the pre-trained neural bug detectors on real bug fixes (to investigate the impact of training on mutants and real bug fixes) or start the training from scratch. An overview of the training process can be found in Fig. 5.1.

Pre-training with code mutants. During the pre-training phase, we largely follow the training process of existing neural bug detectors [HSS+19]. Therefore, our training objective is *not* to identify real bugs but rather to identify and transform mutated code back to its original form. For this, we start with a general corpus of code snippets (e.g. function implementations) that are obtained by mining public Git repositories. Then, we randomly *mutate* each code snippet in our corpus up to k times which produces a dataset of at max k *unique*[1] mutants per code snippet. During our experiments, we both vary the number of mutants generated per code snippet and the employed mutation operator. To obtain examples of unmutated (likely bug-free) code, we employ the original code corpus. Based on the two datasets, the neural bug detector is then trained to (1) distinguish mutated from the real unmutated code, (2) identify the mutation location (if any), and (3) find the code replacements needed to revert the code mutation. As the dataset of mutants is up to k times larger than the original code

[1]We ensure that each generated mutant is unique. As the number of possible single token mutations is limited by the number of tokens and token replacements for a given code snippet, we might generate less than k unique mutants.

corpus, we additionally *supersample* the correct code snippets during training to match their frequency with the mutants. In practice, to achieve this, we implemented a *two-stream* training process that samples correct code snippets at the same frequency as mutants seen during the training. This avoids biasing the model towards the dominant class (e.g. correct code or mutants).

Fine-tuning with real bug fixes. During the fine-tuning phase, we continue (or restart) our training on real bugs and realistic bug-free code. To obtain code containing real bugs, we employ the code related to a real bug fix *before* the fix is applied. For realistic (likely) bug-free code, we again utilize the Github code corpus. During our experiments, we vary the number of real bug fixes employed during training and the size of the code corpus. With the help of real bug fixes, the neural bug detectors is then trained to (1) distinguish *real buggy code* from likely bug-free code, (2) identify the bug location (if necessary) and (3) imitate the original bug fix. To again obtain a balanced distribution during training, we supersample the buggy programs to match the frequency of the bug-free code.

5.3.2 Scaling Factors in the Training of Neural Bug Detectors

There are mainly two scaling factors that might influence the training of neural bug detectors: (1) the scale of the real bug fix dataset used during fine-tuning, (2) the scale of the mutant dataset used during pre-training and (3) the scale of the Github code corpus used both during pre-training and fine-tuning.

Real Bug Fixes. Real bug fixes are typically hard to obtain at a large-scale which has influenced the design and evaluation of existing neural bug detectors [VKM+19, HSS+19, AJFB21]. With `CTSSB-0.9M` and `CSSB-2.3M`, we now have access to two large-scale collections of real bug fixes which we utilize in our study. Still, we are interested whether smaller collections would have been sufficient to boost the performance of neural bug detectors. For this reason, we take several subsamples of our real bug fix dataset ranging from a few hundred examples to a large-scale dataset of multiple thousands real bug fixes. During our study, we evaluate the impact of scaling up the real bug fix dataset during training on the performance of existing neural bug detectors.

Mutants. To scale up the mutant dataset during pre-training, we can either (1) increase the mutation frequency (number k of mutants generated per code snippet) or (2) increase the number of code snippets used for mutation.

Mutation frequency. Existing works often limit themselves to generate $k = 1$ [HBV22], $k = 3$ [HSS+19] or $k = 5$ [AJFB21] mutants per code snippet in the Github code corpus. Hellendoorn et al. [HSS+19] motivated this limitation to avoid biasing the generated dataset too strongly towards longer functions. It is however unclear whether this would hurt the performance of neural bug detectors. Therefore, we are interested

how increasing the number of mutants beyond what is evaluated in previous work during training impacts the performance of neural bug detectors.

Code snippets. Even if we fix the number k of mutants generated per code snippet, we can increase the total number of mutants by increasing the number of code snippets mutated. Although code in open source projects has become abundant, existing work often only focuses on a relatively small subset of what is available online.

Github code corpus. The number of code snippets in the Github code corpus can both impact the pre-training and fine-tuning phase. During pre-training, it both impacts the number of different bug-free code snippets and mutants seen during training. During fine-tuning, the code corpus is used to further train the neural bug detector on bug-free code. Therefore, we investigate the impact of scaling up the Github code corpus on the performance of neural bug detectors.

5.3.3 Implementation

To effectively measure the impact of real bug fixes and mutants at scale on different neural bug detectors, we implemented several neural bug detector baselines in a unified framework. During this process, we largely followed the design of Hellendoorn et al. [HSS+19]: We implemented a common neural bug detector architecture with exchangeable components (e.g. the token encoder, the neural encoder and the localization and repair heads are freely exchangeable). The token encoder encodes each token as a set of subtokens and it averages the subtoken embeddings to obtain a token encoding. The token encodings are processed by different types of neural encoders. For this, we reimplemented or reused state-of-the-art components. The Transformer based neural encoders are built upon the official BERT [DCLT19] implementation from the `transformer` library [WDS+20]. The graph-based neural encoders are implemented closely to the implementation of the PyBugLab model [AJFB21]. For preprocessing the code, we reimplemented the code preprocessing pipelines for tokenization [HSS+19] and graph construction [AJFB21].

5.4 Evaluation

In our experiments, we evaluate the impact of real bug fixes and mutants at different dataset scales on the performance of neural bug detectors for single token bugs in Python. In the process, we aim to answer the following research questions:

RQ1 How does training on real bug fixes at different dataset scales impact the localization and repair performance of neural bug detectors on real bugs?

RQ2 How does training on mutants at different dataset scales and with different types of mutators impact the localization and repair performance of neural bug detectors on real bugs?

RQ3 How do neural bug detectors trained on large scale datasets compare to state-of-the-art neural bug detectors?

For answering the research questions, we investigate the correlation between dataset scale and the performance of neural bug detectors by varying the number of mutants and real bug fixes available during training. As a part of scaling up the mutant dataset in RQ2, we also vary the size of the Github code corpus.

5.4.1 Evaluation Tasks

To answer our research questions, we evaluate the neural bug detectors using the following two tasks: (1) *real bug detection and repair* and (2) *correct code identification*. In the first task, we evaluate the ability of neural bug detectors to detect, localize and repair real bugs found in open source projects. The second task serves to evaluate the ability of neural bug detectors to identify bug-free code.

Real bug detection and repair. For evaluating the ability to detect, localize and repair real bugs, we employ the PyPIBugs benchmark [AJFB21]. PyPIBugs consists of 2374 real world Python bugs and their fixes derived from open source projects. The benchmark is hand-filtered and hence is likely to include bugs that represent real world bugs. During our evaluation, we only consider single token bugs (which excludes argument swaps) in functions where the implementation is still publicly available[2]. This resulted in a real world benchmark consisting of 2028 real world bugs.

Correct code identification. We employ the test split of ETH Py150K [RBV16] for evaluating the ability of the neural bug detector to identify bug-free code. The test split consists of 50K Python files. We extracted all top-level functions and removed all near-duplicates [All19] in the resulting dataset. This process led to a benchmark of more than 200K Python functions. We assume that all the gathered function implementations are bug-free and that any alarm reported by the neural bug detectors are false alarms.

Metrics. To evaluate the effectiveness of the neural bug detectors on the real bug detection and repair task, we measure the *joint recall* of detecting, localizing and repairing real bugs. In addition, we also report the *localization recall* of identifying the bug fix location and the *repair recall* of finding the correct repair. For quantifying the precision of our bug detectors on the correct code identification task, we measure the *false positive rate* (FPR). The false positive rate is the ratio of reported false positives (false alarms) and the number of bug-free code examples.

[2]The benchmark only references the original bug fixing commit. We found that not all commits were publicly available at the time of mining them.

5.4.2 Neural Bug Detector Baseline

We train and evaluate three types of neural bug detectors based on (1) *transformers* [VSP+17] (with relative attention [SUV18]), (2) *graph neural networks* (GNN[3]) [AJFB21] and (3) *graph relational transformers* (GREAT) [HSS+19]. The transformer learns to detect and repair single token bugs directly on the function implementation. The graph based models (GNN and GREAT) learn to localize and repair single token bugs based on structural-, control flow- and data flow information. For a fair comparison, none of the models considered in our experiments are pre-trained, all models share a similar size (25M - 34M parameters) and see the same number of training examples. The models are trained to support code snippets up to 1024 *program* tokens. During our evaluation, we consider all bugs in larger functions as undetected. To support the four bug types, we initialize the vocabulary V to the set of unary and binary operators defined in Python together with the set of boolean and numeric literals ($x \in \{-2, -1, 0, 1, 2\}$).

Training baseline. For evaluating the impact of real bug fixes and mutants, we train the neural bug detectors in different training setups. As a baseline for the training process, we adopt the setup of Hellendoorn et al. [HSS+19]. The neural bug detectors are trained on a training dataset purely consisting of mutants (with $k = 5$) for 300 epochs (a 200K examples per epoch) with early-stopping on the validation set. In our experiments, we vary the number of mutants injected and the number of real bug fixes seen during training.

5.4.3 Datasets

For training the neural bug detectors, we employ two types of datasets: a general *Github code corpus* and a dataset of *real bug fixes*. To achieve comparable results, we focus here also on bugs in Python function implementations.

Github code corpus. As a general corpus of Python code, we consider the train split of the Py150K dataset [RBV16]. The train split consists of around 100K files. We hold out 10K files which we use as a validation set for validating our neural bug detectors. We extract all top level functions and deduplicate the dataset such that functions that do occur in the training process do not occur in our validation and test benchmarks. In total, our training corpus consists of more than 360K Python function implementations.

To train the neural bug detectors, we randomly inject up to k mutants that represent one of our four bug types. For this, we considered two types of mutation operators: (1) *traditional random* and (2) *contextual* mutation operators. The main difference between these mutation operators is that the former injects mutations purely randomly,

[3]Our evaluation setup slightly differs from [AJFB21] in that only function level information are considered. Therefore, any relation that requires access to an implementation outside the scope of a single function cannot be computed.

while the later injects mutations dependent on the surrounding mutation context (see Chapter 3). For our experiments, we evaluated several different types of contextual mutation operators (see Appendix A.2) and selected the contextual mutation operator that is most effective in reproducing real bugs.

Real bug fixes. For obtaining real bug fixes at a sufficient scale, we constructed a novel dataset of 33K real Python bugs for training. As a starting point, we used our `CSSB-2.3M` dataset. We started by pre-filtering the dataset for bug fixes that likely fall into one of our bug categories. We then mined the original repositories for the function implementation corresponding to the bug-fixing code commit. Since not all bug types can be identified purely on the code commit (e.g. a variable misuse requires that all variables are defined in scope), we filtered and deduplicated[4] the dataset of buggy Python functions for a second time. This process has led to around 35K examples of real world Python bugs and their fixes that match at least one of our bug types. During training, we use 33K examples for training and hold out 2K examples for validation.

Scaling Up. To evaluate the impact of scaling, we evaluate neural bug detectors trained on datasets at different sizes. For real bug fixes, we consider several subsets ranging from a few hundred examples to the complete dataset. For mutants, we scale the dataset by increasing the number k of mutants injected. In addition, we also experimented with scaling up the Github corpus used for the mutation process. For this, we collected examples from the Python part of CodeSearchNet [HWG+19] which we deduplicated with respect to our train and test sets. This resulted in an additional 384K examples that we can use to scale up our Github code corpus and our mutant datasets.

5.5 Results

In the following, we discuss our experimental results with the goal of answering our research questions.

5.5.1 RQ1 - Impact of Real Bug Fixes at Scale

To evaluate the impact of real bug fixes at different dataset scales on the training process, we use our real bug fix dataset for training the neural bug detector baselines.

Experimental setup. In our experiments, we both consider neural bug detectors trained purely on real bug fixes and bug detectors fine-tuned from our baseline setup. During this process, we consider subsamples of 0% (no fine-tuning), 1% (334), 3% (996), 5% (1.658), 10% (3.314), 30% (9.936), 50% (16.559), 70% (23.180), 90% (29.802), 100% (33.113) of the real bug fix dataset. We train and evaluate bug detectors for each sample size. To obtain more unbiased results, we train our neural bug detectors on

[4] I.e. we removed all bug fixes that appear in our test sets.

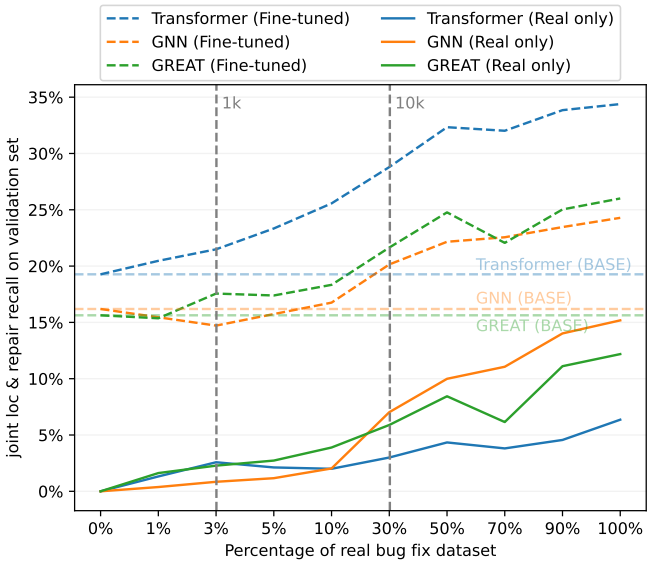


Figure 5.2: Effect of real bug fixes at different dataset scales on the performance on the validation set. The x-axis is the percentage of real bug fixes used during fine-tuning.

three subsamples for each sample size and report the averaged results on our validation set.

Results. The averaged results are shown in Fig. 5.2. We report the joint localization and repair performance on the validation set for both the neural bug detectors trained from scratch (solid line) and the neural bug detectors fine-tuned from our baselines (dashed line). For comparison, we also report the validation performance of neural bug detectors purely trained on mutants (dashed horizontal line). The vertical gray dashed lines mark datasets that exceed 1K and 10K examples, respectively.

Training purely on real bug fixes. We find that neural bug detectors trained purely on real bug fixes underperform in comparison to our baselines. The GNN is the only model that achieves comparable performance when trained on the complete real bug fix dataset. Interestingly, we still find that *scaling up* the number of real bug fixes has a significant impact on the performance of neural bug detectors.

Fine-tuning from baselines. When fine-tuning our neural bug detector baselines with real bug fixes (dashed lines), the performance of the neural bug detectors continues to improve as the size of the real bug fix dataset increases. Even training on small datasets of 1K real bug fixes (less than 5% of the dataset) can significantly improve the performance of neural bug detectors. This is encouraging as datasets of these sizes are often available [KS20]. Still, by scaling up the dataset by a factor of 20 (i.e. from 5% to 100%) can further improve the bug detector performance by up to 152%.

Benchmark performance. To evaluate the impact of training on real bug fixes on our benchmark tasks, we evaluate the best performing neural bug detectors, i.e. those fine-tuned from our baselines on 100% of our dataset, on real bug detection and repair

Table 5.1: Impact of training with real bug fixes on the performance of neural bug detectors on our tasks of real bug detection and correct code identification

	Transformer				**GNN**				**GREAT**			
	FPR↓	Joint↑	Loc.↑	Repair↑	FPR↓	Joint↑	Loc.↑	Repair ↑	FPR↓	Joint↑	Loc.↑	Repair↑
Only real bugs	**12.1**	10.9	14.1	40.4	**14.8**	15.4	22.4	42.7	**10.0**	12.9	16.5	50.1
Only mutants	25.2	21.4	25.8	59.9	26.2	18.4	24.2	53.3	27.2	19.1	23.7	56.2
Mixed	27.0	24.5	29.6	65.5	20.9	18.9	24.6	56.7	27.6	21.7	25.6	60.5
Fine-tuned	26.7	**32.2**	**37.5**	**68.6**	17.4	**24.1**	**29.9**	**59.3**	21.3	**27.0**	**31.7**	**64.4**

and on correct code identification. For comparison, we consider neural bug detectors trained purely on mutants and neural bug detectors trained purely on real bug fixes. In addition, we also consider a neural bug detector that is trained on a mix of real bug fixes and mutants (Mixed). Our evaluation results are shown in Table 5.1. We find that the fine-tuned neural bug detectors significantly outperform all other variants in terms of localization (Loc.), repair (Repair) and joint localization and repair (Joint.) recall. In addition, training on real bug fixes can improve the false positive rate. However, the lowest false positive rate is achieved by the neural bug detectors trained solely on real bug fixes. Training on mutants (in all variants) can increase the false positive rate but more importantly it also boosts the ability of neural bug detectors to detect and repair real bugs significantly.

We conclude for RQ1:

> Scaling up the number of real bug fixes seen during training can significantly boost the performance of neural bug detectors. In comparison to neural bug detectors purely trained on mutants, the fine-tuned neural bug detectors improve the ability to detect and repair real bugs by up to 150%. They still maintain a comparable false positive rate on correct programs.

5.5.2 RQ2 - Impact of Mutants at Scale

While scaling up the real bug fix dataset is often challenging, the number of mutants seen during training can be scaled up more easily. For this, we employ two strategies: we either (1) increase the number k of mutants generated for each code snippet (mutation frequency) or (2) we increasing the size of the code corpus used for mutation.

Experimental setup. To evaluate the impact of mutants at different dataset scales, we trained and evaluated several different neural bug detectors that mainly differ in the number of distinct[5] mutants seen during pre-training. For this, we performed two types of experiments: First, we vary the mutation frequency between 1, 3, 5, 10, 100 and 1000 unique mutants generated per code snippets. Second, we increase the number

[5]As we fix the training to 300 epochs (a 200K examples per epoch), the number of mutants in each training run is the same (60M mutants and bug-free code snippets). However, the number of distinct training examples might be significantly less and therefore the same mutant might be repeatably seen during training. We can mitigate this by increasing the number of distinct mutants in our training dataset.

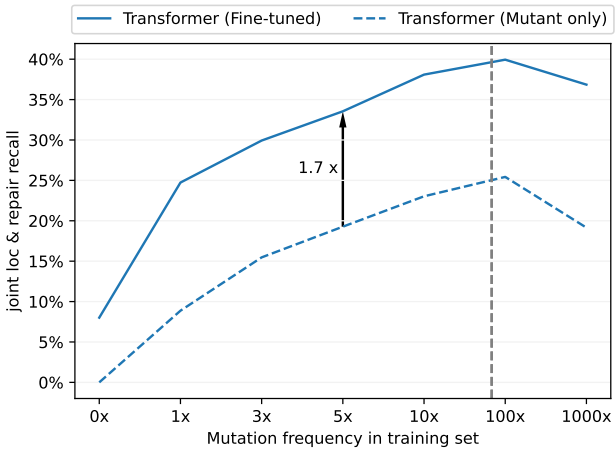


Figure 5.3: Effect of mutation frequency during training on the performance of Transformer-based neural bug detector on the validation set. The gray dashed line represents the average number of unique mutants that can be generated per code snippet.

of distinct mutants by increasing the Github code corpus. For this, we include more than 384K function implementation from CodeSearchNet in our general Github corpus and use this as starting point for mutation. During mutant generation, we vary the type of mutation operator used for mutation. For each generated dataset, we pre-train the neural bug detectors and evaluate their performance before and after fine-tuning on real bug fixes.

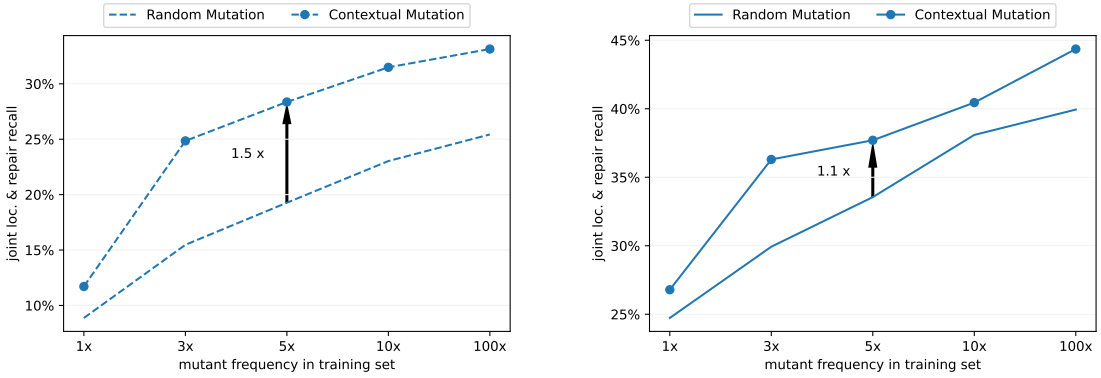
Results. Fig. 5.3 and Fig. 5.4 present overviews of our results for the first set of our experiments. Here, we measure the joint localization and repair recall on our validation set for the neural bug detectors trained with different mutation frequencies. The configuration 0x represents a version of the neural bug detector trained only on real bug fixes.

Impact of higher mutation frequencies. For all evaluated neural bug detectors, we observe that increasing the mutation frequency (up to a critical point of 100x mutants per code snippet) leads to a performance improvement for both localization and repair. This is surprising as the number of unique mutants per code snippet is limited (with an average of 85 unique mutants per code snippet) and hence longer functions with more mutant candidates are oversampled. However, increasing the mutation frequency beyond 100x does not always lead to improvements.

Impact on fine-tuning performance. We find that fine-tuning on real bug fixes has an orthogonal effect on the localization and repair performance of the neural bug detectors. By fine-tuning on real bug fixes, we can boost the performance of the neural bug detectors by 1.5x up to 1.7x. The effect of scaling up the mutant dataset also transfers to the performance of the fine-tuned neural bug detectors. By scaling up the mutation frequency from 5x to 100x, we can increase the performance of the fine-tuned models

(a) GNN (b) GREAT

Figure 5.4: Effect of mutation frequency during training on the performance of the graph-based neural bug detectors on the validation set. The gray dashed line represents the average number of unique mutants that can be generated per code snippet.



(a) Before Fine-tuning (b) After Fine-tuning

Figure 5.5: Effect of using a more realistic mutator on the performance of the Transformer-based neural bug detector on the validation set.

by 1.16x (Transformer) up to 1.4x (GREAT).

Impact of more realistic mutations. We evaluate the impact of scaling up the number of mutants with a more realistic mutator. For this, we employ a contextual mutator at different mutation frequencies. We compare the performance of neural bug detectors trained on more realistic mutations to those trained on traditional random mutations. Our results for the Transformer based neural bug detector are shown in Fig. 5.5. We find that our results are consistent with our findings in Chapter 3: Training on more realistic mutants can significantly boost the (joint) performance of a Transformer-based neural bug detector (by up to 1.5x). Scaling up the mutation frequency can further boost the performance of a Transformer-based neural bug detector (even when using more realistic mutations). Interestingly, we find that using more realistic mutations has a significantly smaller effect on the fine-tuning performance (from around 1.5x before fine-tuning to 1.1x after fine-tuning).

Table 5.2: Evaluation results for the improved neural bug detectors on our benchmark tasks.

	Transformer				**GNN**				**GREAT**			
	FPR↓	Joint↑	Loc.↑	Repair↑	FPR↓	Joint↑	Loc.↑	Repair ↑	FPR↓	Joint↑	Loc.↑	Repair↑
Baseline	25.2	21.4	25.8	59.9	26.2	18.4	24.2	53.3	27.2	19.1	23.7	56.2
+ 100x Mutants	30.0	24.9	30.4	65.6	20.4	18.7	24.2	58.3	28.2	23.8	29.0	63.9
+ Improved Mutator	25.7	27.6	33.5	64.1	24.6	19.1	24.7	57.8	26.6	23.5	29.7	61.1
+ Real Bug Fixes	20.7	37.6	42.9	73.4	19.9	**26.5**	**31.9**	65.9	21.7	34.4	38.7	71.7
+ Larger Corpus	**16.1**	**38.6**	**43.2**	**75.1**	**15.2**	26.2	30.6	**67.8**	**16.2**	**35.1**	**39.1**	**74.9**

Benchmark performance. We evaluate the impact of increasing the mutation frequency (to 100x) and employing the improved mutation operator during training on the benchmark performance. Results are shown in Table 5.2. We find that pre-training on a larger set of mutants significantly improves the detection and repair of real bugs across all neural bug detectors. However, increasing the mutation frequency comes also at a cost of increasing the false positive rate (for Transformer-based and GREAT-based neural bug detectors). Pre-training on mutants generated by our improved contextual mutation operator can mitigate the problem while boosting the ability of the neural bug detector to detect real bugs. However, the repair performance also slightly decreases across all neural bug detectors when switching to the improved contextual mutation operator. Interestingly, we find that improving the pre-training process also transfers to the performance after fine-tuning: When pre-training on a higher number of more realistic mutants, the joint localization and repair recall of the fine-tuned neural bug detectors increases by up to 1.27x (in comparison to the baseline fine-tuned on real bug fixes; see Table 5.1) and the false positive rate decreases by a factor of up to 0.78x.

Impact of larger Github corpora. As the improvement in performance due to an increase in mutation frequency is limited (and stops after 100x), we also consider increasing the Github code corpus. For this, we included the Python part of the CodeSearchNet corpus in our Github code corpus and used the joint corpus as a starting point for the mutation process. We again generate up to 100 mutants per code snippet and pre-trained the neural bug detectors on the new datasets. Our results are shown in the lower part of Table 5.2. We find that increasing the size of the Github code corpus can slightly improve the ability of the neural bug detectors to detect and fix real bugs. Interestingly, the size of the Github code corpus has a significant impact on the false positive rate which further decreases by a factor of up to 0.75x. We expect that further scaling up the Github code corpus can yield further improvements[6]. We leave this evaluation open for future work.

Finally, we conclude for RQ2:

[6]We already experimented with a larger code corpus containing more than 90M function implementations. However, first experiments on the code corpus did not yield significant improvements. Therefore, we believe that advanced data filtering and novel ways to balance corpus size and mutation frequency are needed to fully exploit code corpora of this size.

Table 5.3: Comparison with PyBugLab on the PyPIBugs benchmark.

Bug type	**PyBugLab [AJFB21]**		**Transformer (ours)**		– Real Bug Fixes	
	Loc.	Repair	Loc.	Repair	Loc.	Repair
Wrong Assign Op	20.0	68.9	**43.2**	**77.3**	36.4	77.3
Wrong Binary Op	27.2	54.3	**43.6**	**83.1**	16.9	64.8
Wrong Boolean Op	27.6	**96.9**	**34.1**	95.6	4.9	76.4
Wrong Comparison Op	33.7	**66.1**	**42.6**	65.3	22.7	60.3
Wrong Literal	21.6	**78.4**	**33.7**	78.0	17.4	83.7
Variable Misuse	35.3	70.5	**45.4**	**74.7**	42.0	75.0

Pre-training on mutants can significantly boost the ability of neural bug detectors to detect and repair real bugs. Increasing the number and realism of the mutants seen during training are both effective strategies to further boost the performance (by up to 130%). However, a side effect of pre-training on mutants is a higher false positive rate which can only be partly mitigated by training on more realistic mutants and by increasing the size of the code corpus.

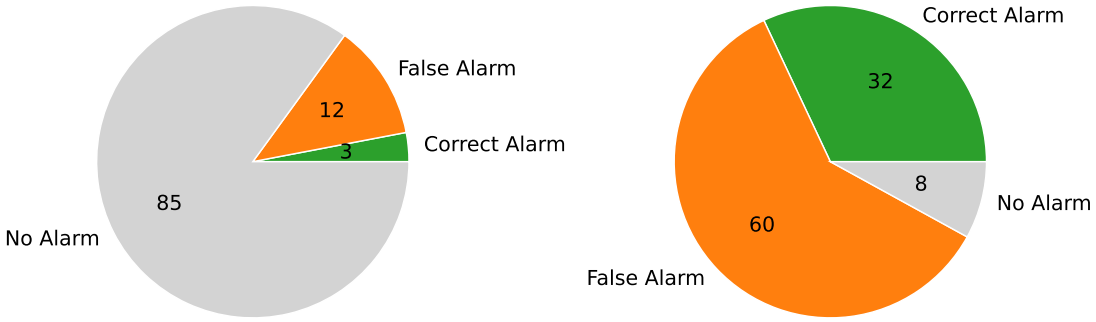
5.5.3 RQ3 - Comparison with State of the Art

After having evaluated the impact of real bug fixes and mutants on the performance of neural bug detectors, we are now interested how our best performing neural bug detector[7] compares with state-of-the-art approaches.

Comparison to PyBugLab [AJFB21]. We evaluate the best performing neural bug detector against PyBugLab [AJFB21] on our real bugs benchmark (PyPIBugs). In comparison to our neural bug detectors, PyBugLab employs a different (adversial) mutation operator for training and uses additional augmentations of the training data. PyBugLab does not employ real bug fixes during training. Unfortunately, the PyBugLab neural bug detectors are *not* publicly available. Therefore, for our comparison, we rely on the numbers reported for each bug type of the original author. The evaluation results are shown in Table 5.3. We additionally report the performance of our neural bug detector before being fine-tuned on real bug fixes. Overall, we find that:

Training on real bug fixes significantly improves the bug localization performance. We find that our neural bug detector has a significantly higher bug localization performance (Loc.) than PyBugLab. Across nearly all bug types, our neural bug detectors localizes (and repairs) more than 10% more real bugs. At the same time, our neural bug detectors achieves a similar or significantly higher repair performance (Repair) for the individual bug types. Key to this performance improvement is the access to a large real bug fix dataset that we can use to fine-tune the neural bug detectors. By fine-tuning the neural bug detector on real bug fixes, the neural bug detector becomes significantly more

[7] According to our results, the best performing neural bug detector is a Transformer-based neural bug detector trained with a larger corpus with 100x improved mutants injected and fine-tuned on real bug fixes.



(a) CuBERT (b) Transformer (ours)

Figure 5.6: Comparison of CuBERT and our neural bug detector on Real Python projects with variable misuse bugs.

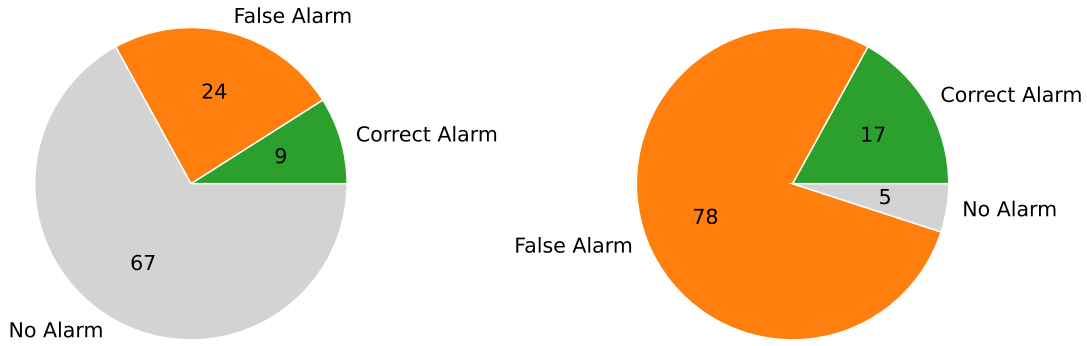
effective in localizing and repairing real bugs for all bug types. Fine-tuning PyBugLab with real bug fixes could further boost its performance and PyBugLab trained on real bug fixes might be more effective for Wrong Binary Op, Wrong Boolean Op, Wrong Comparison Op and Wrong Literal bugs.

Comparison to He et al. [HBV22]. We also compare our neural bug detector with CuBERT [KMBS20], a recent 340M parameters masked language model, fine-tuned by He et al. [HBV22] for the detection and repair of simple bugs. CuBERT comes in three variants, each fine-tuned specifically for variable misuse, binary operator and argument swap bug detection. We focus here on the variants fine-tuned for variable misuses and binary operator bugs as these objectives overlap with ours. For the comparison, we employ our 28M parameters Transformer-based neural bug detector. We configure our neural bug detector so that it only reports variable misuses or binary operator bugs, depending on which variant of CuBERT we are comparing with.

Comparison on PyPIBugs. We compare the two neural bug detectors on subsets of PyPIBugs that tackle variable misuses and binary operator bugs respectively. Despite the size difference, we find that our neural bug detector significantly outperforms CuBERT on PyPIBugs: Here, CuBERT detects and repairs around 2.82% of all variable misuses and 2.23% of all binary operator bugs. Our neural bug detector detects 16x more variable misuses and 14x more binary operator bugs. We however have to note that CuBERT is tuned for *precision* and we expect[8] that CuBERT performs significantly better on the task of correct code identification.

Comparison on Real Python Projects. To get a better understanding of how the two neural bug detectors compare in a more realistic setting, we evaluate them on the task of finding and fixing real bugs in real Python projects. For this, we sampled around

[8]We cannot make a meaningful comparison between our neural bug detectors and CuBERT as the training set of CuBERT and our test set for evaluating correct code identification significantly overlap.



(a) CuBERT (b) Transformer (ours)

Figure 5.7: Comparison of CuBERT and our neural bug detector on Real Python projects with binary operator bugs.

100 variable misuse and 100 binary operator bug fixes collected in PyPIBugs. Each bug fix corresponds to a bug and its fix found in a Python project. Therefore, to create our benchmark, we downloaded the original project and reinserted the original bug. The goal of the neural bug detector is then to rediscover and fix the original bug by scanning all top-level functions of the given project. During this process, the neural bug detector might report multiple functions as buggy. As in practice a developer might not be willing to review all raised alarms [KXLL16], we sort them by the confidence of the neural bug detector, i.e. we sort by $P_\varphi(\langle l, r\rangle \mid T)$, and only report the top-5 alarms per project.

Results. Our experimental results for variable misuse bugs are shown in Fig. 5.6 and for binary operator bugs are shown in Fig. 5.7. The figures show the number of projects with at least one (1) *Correct Alarm*: the original bug is discovered and fixed, (2) *False Alarm*: an alarm is raised but the original bug is not detected and (3) *No Alarm*: the rest. Overall, we find that:

CuBERT is significantly more precise but finds fewer bugs. In comparison to our neural bug detector, CuBERT is significantly more precise. In total, CuBERT reports 22 potential variable misuses and 110 potential binary operator bugs across all projects. CuBERT detects and fixes 3 historic variable misuses and 9 binary operator bugs respectively. In contrast, our neural bug detector fixes 29 additional variable misuses and 8 additional binary operator bugs[9] which CuBERT does not detect but also produces thousands of false alarms more across all projects. At project level, CuBERT reports no alarm for 85 and 67 projects with a known variable misuse or binary operator bug respectively. Even though CuBERT is more precise, it reports a false alarm for 12 projects with a known variable misuse and 24 projects with a known binary operator

[9]In fact, the neural bug detector finds and fixes in total 21 additional binary operator bugs that CuBERT does not detect. However, 13 correct alarms are not considered here as they do not appear in the top-5 alarms per project.

Table 5.4: Example of a Software Bug only found after training on Real Bugs. Code is reformatted to fit the figure.

Example	Description
1 # Assignment Operator Bug 2 if event == 'highstate-start': 3 minions += set(data['minions']) 4 elif event == 'highstate': 5 minions.discard(data['minion'])	The set union in Python is typically computed via the `|=` operator. However, the developer mistakenly uses the `+=` to compute the union of two sets. **Fix:** replace `+=` in Line 3 by `|=`.

Table 5.5: Example of a Software Bug that is not found by any neural bug detector. Code is reformatted to fit the figure.

Example	Description
1 # Variable Misuse 2 def _find_completion(fuser, relation): 3 for fuser_relation in fuser.fusion_graph. → get_relations([...]): 4 if fuser_relation._id == relation._id: 5 return fuser.complete(fuser_relation) 6 return None	A variable misuse bug where the developer uses the variable `fuser_relation` instead of `relation`. The context is insufficient to decide that the developer intends to use the `relation` variable. **Fix:** replace `fuser_relation` in Line 5 by `relation`.

bug.

In the end, it is highly dependent on the application scenario whether a low false positive rate (CuBERT) or a high bug detection and repair performance (ours) is preferred. We can for example envision that our neural bug detectors are highly useful in scenarios where false repairs can be easily rejected via patch validation techniques [YZLT17a] or by elimination techniques for false positives [KMJ+22a, BFHORQ10]. In fact, in Chapter 6, we develop a validation approach that allows us to increase the precision of our neural bug detectors. In this case, having access to a high-recall neural bug detector that detects and repairs a significant number of real bugs can be beneficial.

Overall, we conclude for **RQ3**:

> In comparison to previous neural bug detectors, our neural bug detector can detect and repair significantly more real bugs. In comparison to neural bug detectors tuned for precision, our neural bug detector produces a significantly higher number of false alarms. However, this could be addressed in future work by employing post hoc validation strategies.

5.6 Discussion

We now take a more qualitative look on our experimental results. For this, we manually reviewed the raised warnings on our real bugs benchmark.

Impact of Real Bug Fixes. While neural bug detectors trained on mutants can only exploit how developers implement code (and find deviations), neural bug detectors fine-tuned on real bug fixes can also exploit how developers make mistakes and how they fix them. As a result, as we have seen in RQ1 and RQ2, neural bug detectors trained on

real bug fixes are often more effective in finding and fixing real bugs. When analyzing the bugs that can only be detected and fixed by neural bug detectors trained on real bug fixes, we find that the difference is most visible in the following two cases: for (1) *type-related* bugs and (2) for bugs occurring in code snippets with *multiple* weaknesses (e.g. a real bug that co-occurs with other bugs in the same code snippet and code quality issues at multiple locations). An example of a type-related bug that can only be detected and fixed by neural bug detectors trained on real bug fixes is shown in Table 5.4. Here, the developer confuses the `|=` operator with the `+=` operator to compute a set union. The neural bug detectors catches the bug after being fine-tuned on real bug fixes as it appears more frequently in our real bug fix dataset.

Impact of Real Bug Fixes at a Larger Scale. Some real bugs that occur often during development are less likely to appear in code repositories. For example, the type-related bug in Table 5.4 is less likely to appear in a real code repository. The reason is not that it appears less likely in practice, but because it is a type-related bug, it is more likely to be detected during execution. Still, a good neural bug detector should warn the implementer before it is submitted to a code repository. We believe that our neural bug detectors are able to detect these bugs because they have sufficient *support* in our large scale bug fix collections. Smaller bug fix collections such as the 872 wrong binary operator bugs used in previous work [HBV22] do not include this type of bug. Even larger collections such as PySStuBs [KPBH21] only collect two instances related to set operator bugs. Our training dataset includes more than 20x more bug fixes that match the same bug pattern. Therefore, by increasing the scale of the real bug fix dataset, we can capture unique bug types that appear less frequently in code repositories.

Impact of Mutants at a Larger Scale. Complementary to real bug fixes, our evaluation results suggest that training on mutants at a larger scale can have a significant impact on the ability of neural bug detectors to detect and fix real bugs. While mutants are often not representative for real bugs, a (random) mutation operator can still increase the *support* of real bugs in the training distribution that are otherwise infrequent in code repositories. To demonstrate this, we analyze which real bugs in our PyPIBugs benchmark can be reproduced by our mutation strategies. For this, we mutate the code after the bug fix and check whether the original bug can be reproduced within the first k mutations. For traditional random mutations, we report the worst-case performance (i.e. the original bug is always sampled last) as a lower bound and, for the improved contextual mutation operator, we order mutants according to likelihood of generating them. Results are shown in Fig. 5.8. We find that almost all real bugs can be reproduced when we reach a mutation frequency of 1000x mutants (even if we are "unlucky" and sample the original bug last). Therefore, if a real bug can be reproduced by mutation, increasing the mutation frequency is enough to ensure that the bug is generated via mutation. However, increasing the mutation frequency also

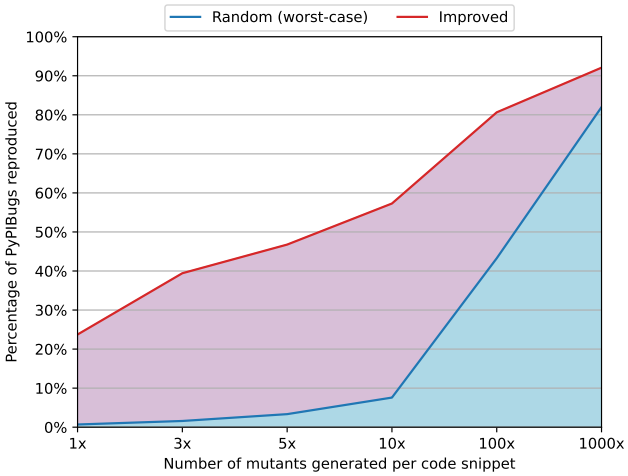


Figure 5.8: Inversion test on PyPIBugs for reproducing real bugs via mutation. The x-axis is the number of mutants generated and the y-axis is the percentage of real bugs reproduced within k mutations. For the traditional random mutation, we report worst-case performance (i.e. the original bug is always sampled last) and for the improved contextual mutator we rank the mutation according to the likelihood of generating them.

has the side effect of a lower signal-to-noise ratio (i.e. the number of real bugs to the number of uninformative mutants). Therefore, as our experiments suggest, fine-tuning on real bug fixes is necessary to utilize the learned bug patterns to detect real bugs.

Failure cases. While our improved training process significantly improved the bug detection capabilities of the neural bug detectors, they still fail to recognize a significant portion of real bugs. To understand why this happens, we analyze common failure cases. A key problem is that the neural bug detectors considered in this work are limited by the available context (consisting of the implementation of a single function). Therefore, bugs that require knowledge about implementations which are not available in context can often not be detected. An example is shown in Table 5.5. Here, the developer intends to complete a given `relation` but instead uses the variable `fuser_relation`. The context is insufficient to determine that completing `relation` is preferred over `fuser_relation`. While this can be partly mitigated by incorporating further context, the detection of bugs requires *explicit* knowledge about the functionality of the implementation which is not always provided by the developer. Therefore, we find that neural bug detectors are most effective for well-documented, idiomatic code.

5.7 Threats to Validity

In the following, we discuss potential threats to the validity of our experiments. We differentiate between external and internal validity.

External validity. In our experiments, we focused on neural bug detectors for single

token bugs in Python. While our training approach is general enough to be applied in various setups, the impact of training with real bug fixes and mutants might not generalize to other programming languages, bug types or neural bug detectors.

Other Bug Types and Programming Languages. As we focus on our experiments on four common bug types in Python, our evaluation results might not generalize to other bug types or other programming languages. To still achieve representative results, we selected bug types that are frequent for Python and are commonly explored by previous studies on neural bug detectors. Still, our experimental results might not generalize to other bug types that are more frequent in other programming languages.

Generalization to Other Benchmarks. We evaluated our neural bug detectors on one of the largest benchmark for real bug detection and repair, PyPIBugs. The benchmark is manually validated and collects bug fixes from thousands of repositories. Still, the performance of the neural bug detectors might not generalize to other bugs that are collected in different (smaller) benchmarks or do not occur in benchmarks at all.

Design of Neural Bug Detectors. The design of our neural bug detectors baseline might have significantly influenced the evaluation results. While we considered three different types of neural bug detectors (based on Transformers, GNNs and GREAT), our experimental results might not generalize to future neural bug detectors that utilize different architectures.

Internal validity. Although a variety of neural bug detectors have been developed in recent years, there does not exist a universally accepted setup for training and benchmarking. Therefore, even though we implemented our baselines as close as possible to the reference implementations, the resulting neural bug detectors might behave differently than in the setup they are originally developed for.

Procedural Bias. To achieve comparable results, we designed our evaluation to match the design of prior studies [HSS+19] on neural bug detectors as close as possible. For example, in our main experiments, we adopted the same publicly accessible Github corpus ETH Py150K, a similar architectural design and similar baselines as employed by Hellendoorn et al. [HSS+19]. To further support a wider range of bug types, we adjusted the architecture and mutation process similar to Allamanis et al. [AJFB21]. Still, our evaluation results for the baseline algorithms are slightly different than in previous studies. For example, we found that in contrast to Allamanis et al. [AJFB21] the graph-based neural bug detectors (GNN and GREAT) underperform in our evaluation setup. We attribute this to two main differences in our evaluation setup: First, for a fair comparison, all models are limited to the function implementation without having access to other inter-procedural implementation details. This also prohibits the computation of type related or call structure related information exploited by the graph-based neural bug detectors in Allamanis et al. [AJFB21]. Second, all neural bug detectors were trained on a different (potentially smaller) dataset. Although integrat-

ing this type of information and training on larger datasets would potentially benefit all bug detector baselines, the performance ranking between bug detectors might differ.

Implementation Bias. A second reason for the difference in performance might lie in the implementation details of the neural bug detectors. Even though we implemented the neural bug detectors as close as possible to the original design, there still exist differences in implementation details that might have led to different evaluation outcomes. For example, we found that while graph-based neural bug detectors still perform better in a low data regime (e.g. when trained on real bug fixes), Transformers show a surprisingly strong performance when pre-trained on mutants. We mainly attribute this difference in performance to our choice of a *relative* attention mechanism over an *absolute* attention mechanism used in prior works. In our preliminary experiments, this led to a boost of up to 20% on our validation dataset. Note however that Hellendoorn et al. [HSS+19] already anticipated this result when the models are trained for longer durations. In our experiments, we train on approximately 2.4x more training examples than Hellendoorn et al.

Impact of Randomness. While we designed our experiments to be replicable, there still exists two sources of (simulated) randomness that might have impacted our evaluation results. To evaluate the impact of real bug fixes at scale on the training process, we randomly sampled several subsets from our real bug fix datasets. Therefore, training the neural bug detectors on different subsets might lead to different evaluation results. To mitigate this problem, we trained all neural bug detectors on different subsamples and reported the average results across subsamples of the same size. A second source of randomness is the mutation process. While mutation is an effective way for generating training examples for neural bug detectors, the mutation process itself is inherently random. Therefore, applying the same mutation operator on the same code bases twice might result in two completely different training datasets. To still enable the replication of our experiments, we fix the random seed during mutation which allows to (deterministically) reproduce our training datasets. Still, employing a different randomization process or a different random seed might generate a different training set which in turn might impact the performance of the neural bug detectors.

5.8 Related Work

In this chapter, we have systematically investigated the impact of the size of mutant and real bug fix datasets used during training on the performance of neural bug detectors. In the following, we discuss the most closest related work that also addresses the training of neural bug detectors. In addition, we also compare with approaches that exploit real bug fixes for the training of automatic repair or code mutations and we consider alternative pre-train-and-fine-tune techniques.

Training Neural Bug Detectors. Neural bug detectors for detecting and repairing single

token bugs have been explored in previous work [ABK17, VKM+19, HSS+19, AJFB21]. Allamanis et al. [ABK17] addressed the detection and repair of variable misuse bugs by modeling programs as graphs. Vasic et al. [VKM+19] proposed a joint model for the same task and Hellendoorn et al. [HSS+19] explored alternative neural bug detector architectures. All these techniques share a very similar training recipe: they are trained solely on artificially generated mutants which are introduced by mutating existing source code. In contrast, we showed that the integration of real bug fixes (especially at a large scale) in the training process can significantly boost the performance of neural bug detectors. More recent works [AJFB21, PP21, AHO24] have investigated the effect of training data quality on the performance of neural bug detectors. For example, both employing more realistic mutations (as we have shown in Chapter 3) and mutations that are hard to detect [AJFB21] during training can boost the localization and repair performance of neural bug detectors. However, integrating these approaches often increases the complexity of the training process by learning the mutator either prior or concurrently to the training. We found that the integration of real bug fixes in the training process while relying on simpler and easier-to-implement mutation operators that we employ at a large scale can already be sufficient to achieve a significant improvement in the real bug detection and repair performance. However, our experiments also provide evidence that training on more realistic mutations can still boost the performance (even if real bug fixes are available during training). He et al. [HBV22] investigated the effects of training neural bug detectors with a more realistic training distribution. Although their method significantly improved the *precision* of existing neural bug detectors, the resulting neural bug detectors are still limited in their ability to detect and repair real bugs. We found that increasing the scale of the real bug fix and mutant dataset can significantly improve the performance of neural bug detectors *while* training on real bug fixes can also decrease the false positive rate. Bui et al. [BWH22b] found in a subsequent study that pre-trained large language models can be turned into effective neural bug detectors by fine-tuning directly on real bug fixes (without first pre-training on mutants). As our study shows that training on mutants can significantly boost the real bug detection and repair performance, it would be interesting to investigate whether mutants have the same effect on pre-trained language models. We leave this open for future work.

Real bug fixes in the training of automatic program repair techniques. While code mutants have been predominantly used in the training of neural bug detectors, real bug fixes have a long history in the training of automatic program repair (APR) systems [TWB+19b, CKT+21, LWN20, LPP+20b, BSPC19b, CDAR20, GPKS17, LAR17, TPW+19, YMM22, ZSX+21, JLT21b, SFM23, YM24]. For example, SequenceR [CKT+21] learns from thousands of real bug fixes to predict one-line patches. Dlfix [LWN20] and CoCoNuT [LPP+20b] improved the repair performance by proposing a learning strategy that better exploits real bug fixes. In contrast, in our experiments, we evaluated

the impact of training on real bug fixes on the bug detection *and* repair performance. We found that training on real bug fixes significantly improve the ability of neural bug detectors to detect and repair real bugs. Interestingly, existing APR systems seldomly utilize code mutants during training. We however found that training on code mutants can significantly improve the repair performance. This observation is also supported by DrRepair [YL20], SelfAPR [YML+22] and ITER [YM24] which use artificially generated mutants to pre-train APR systems.

Real bug fixes in the training of code mutation operators. Code mutation addresses the inverse problem of injecting realistic bugs into correct code. While most existing code mutation operators are heuristically defined, Tufano et al. [TWB+19a], Patra and Pradel [PP21], and Nong et al. [NOP+23] showed that real bug fixes can be effectively utilized to train code mutation operators. Yasunga et al. [YL21] showed that real bug fixes can be used to train a breaker and fixer that then teach each other to inject and fix realistic syntactic bugs. While our experiments mainly showed that real bug fixes are crucial for bug detection, we found in Appendix A.2 that real bug fixes can be effectively exploited in the mutation process.

Pre-training and fine-tuning. Pre-training on a large corpora and then fine-tuning for a specific task at hand with a smaller dataset has been shown to be highly effective in domains such as natural language processing [DCLT19], image processing [KBZ+20] and mostly recently in programming language processing [FGT+20, KMBS20]. Existing works often pre-train on a generic unrelated task where data is available at a large scale before fine-tuning the model on a specific task. In our evaluation, we directly pre-train the neural bug detectors on the task of identifying and repairing (mutated) code. Kanade et al. [KMBS20] showed that a general pre-training on Python code can improve the detection performance on variable misuses. The model is however fine-tuned on code mutants instead of real bug fixes. In our evaluation, we however found that real bug fixes in the training are crucial for high performing neural bug detectors. Therefore, a combination of these two approaches could be interesting and we leave this open for future work.

5.9 Contributions and Conclusion

In this chapter, we investigated the impact of real bug fixes and mutants at scale during training on the performance of existing neural bug detectors. For this, we systematically evaluated and compared neural bug detectors trained on mutants, real bug fixes and mixtures thereof at different dataset scales. We found that existing neural bug detectors can both utilize mutants *and* real bug fixes when we first pre-train the neural bug detector on a large set of mutants and then fine-tune it on real bug fixes. Our evaluation on thousands of real Python bugs shows that existing neural bug detectors were limited by the available training data. Scaling up both the number of real bug

fixes and mutants seen during training can significantly boost the ability of neural bug detectors to detect and repair real bugs. We believe that our neural bug detectors can be highly useful in scenarios where detecting and fixing real bugs has a higher priority than avoiding false alarms. For example, Dinh et al. [DZT+23] showed that our neural bug detectors can be used to improve the performance of large language models on tasks of code completion with potential bugs.

Limitation. Although we achieved promising results on the task of real bug detection and repair by training on large scale datasets of real bug fixes and mutants, the usability of the resulting neural bug detectors might still be limited by their high false alarm rate. Studies on the usability of static analyzers [JSMB13, SAE+18] suggest that while static analyzers can be useful they are only effective if they do not flood the user with false alarms. We believe that this could also be a potential limitation for the application of neural bug detectors. To mitigate this problem, we investigate in the following chapter whether the output of neural bug detectors can be *validated* with the help of large language models. Our goal is to reduce the number of false alarms while maintaining high bug detection and repair performance.

6 False Alarm Reduction

While in the previous chapters we mainly focused on increasing the ability of neural bug detectors to detect and repair real bugs, we now turn to another problem that might limit the usability of neural bug detectors in practice: *the high number of false alarms*. Recent studies [JSMB13, SAE+18] have shown that even the most effective analyses become quickly impractical if they overwhelm the user with too many false alarms. Therefore, to mitigate this problem for neural bug detection, we propose a *post-hoc* validation strategy that utilizes existing large language models and additional context typically not available to the neural bug detector to detect and reject false alarms. As a result, we are able to significantly reduce the number of false alarms while maintaining the ability of neural bug detectors to detect and repair real bugs.

6.1 Motivation

Finding and fixing software bugs remains to be one of the central challenges of software development [AL21]. Software developers often spent a considerable part of their time debugging. Static bug detectors such as SpotBugs [HP04] but also existing neural bug detectors could significantly speed up the debugging process by finding bugs faster and earlier in the development process where they are still easy to fix [AHM+08]. However, existing tools are seldom used in practice [AHM+08]. Recent studies [JSMB13, BBC+10, SAE+18] on static bug detectors suggest that a key reason against their adoption is the fact that they raise too many *false alarms*. Users are often easily overwhelmed by the sheer number of alarms raised. As only a few of the alarms relate to real bugs [SFZ11], users are often more likely to ignore the raised alarms even if this means that they miss potential bugs.

Motivated by this finding, we aim to address the problem of false alarms in the context of neural bug detection. Neural bug detectors have become increasingly effective in finding and fixing real bugs [ABK17, VKM+19, HSS+19, AJFB21] (as we have seen in

Chapter 5). However, similar to existing static bug detectors, most neural bug detectors suffer from a high false alarm rate [AJFB21] which might limit their usability. One solution [HBV22] is to train the neural bug detector explicitly to avoid false alarms. While this significantly reduces the number of false alarms raised by the neural bug detector, it also significantly impairs its ability to detect and fix real bugs.

Therefore, our goal in this chapter is to reduce the number of false alarms while maintaining the ability of the neural bug detector to detect real bugs as well as possible. For this, we hypothesize that existing neural bug detectors are limited by the available context as they are often trained for *function-level* neural bug detection [AJFB21]. As a result, they might make mistakes that would have been otherwise avoidable with more context.

An approach that would likely mitigate this problem would be to directly train on a longer file-level context. However, training neural models (which are the key component of neural bug detectors) on longer contexts is often prohibitively expensive [CQT+24] and an active area of research [LLH+24]. In particular, training neural bug detectors beyond a certain context length might require specialized architectures or training techniques that are designed for longer context training [LLH+24, MFG24].

Therefore, to still be able to evaluate our hypothesis *without* extra training, we propose to employ an *external* validator that can make use of additional context for validating the alarms raised by a neural bug detector. The key idea of our validation strategy is to view the problem of validating the output of a neural bug detector as a *patch validation problem* [QLAR15]: As neural bug detectors often relate each raised alarm with a potential bug-fixing patch, we can validate the corresponding patch to detect false alarms. We adopt a technique that already has been proven effective in the context of automatic program repair [YKH+24]: We employ the entropy computed by existing large language models (LLM) [BMR+20] to score potential patches. LLMs are trained on millions of programs with the objective of completing partial code. As bugs are scarce, they often learn to assign a higher probability to fixed code than to its buggy variants [RHG+16]. We exploit this observation to only accept patches where the code after applying the patch is assigned a higher probability than the code before applying the patch. We consider all other patches as potential false alarms. During our experiments, we evaluate our validation strategy in the context of neural bug detection, and we specifically evaluate the impact of using extra file-level context (which is supported by the considered LLMs) for validation. Based on our results, we are able to significantly reduce the number of false alarms while maintaining the ability of neural bug detectors to detect and repair real bugs.

Example. To further motivate our hypothesis, we review an example where the neural bug detector reports a false alarm *because* the available context is insufficient. The example together with the proposed patch is shown in Table 6.1. Here, we executed the Transformer based neural bug detector from Chapter 5 on the recently proposed

Table 6.1: Example of a false alarm raised for projects in DyPyBench [BKP24]

Example	Description
1 `def __gt__(self, other: str) -> bool:` 2 `  return prec(self) < prec(other)` 3 `...` 4 `def __lt__(self, other: str) -> bool:` 5 `- return prec(self) > prec(other)` 6 `+ return prec(self) < prec(other)` 7 `...` 8 `...` 9 `...`	A false alarm that seems reasonable if only the function implementation is given. Restricted to the implementation of `__lt__` function, the neural bug detector wrongly infers that a lesser operator (`<`) to compare objects is more likely in the context of the function. While this seems reasonable at first sight, it becomes clear from the surrounding context (gray lines) that a greater operator is indeed needed (`>`). Existing neural bug detectors typically do not have access to this type of context.

DyPyBench benchmark [BKP24] and collected all false alarms. DyPyBench collects around 50 Python projects with available test suites. We used the available tests to identify *faulty* patches corresponding to false alarms. In this example, we find that the prediction of the neural bug detector seems reasonable at first if we only consider the function implementation. However, as soon as we review the remaining context (given by the gray lines), we find that this patch actually corresponds to a false alarm (which is confirmed by the test suite). In the following, we aim to employ LLMs to perform a similar type of review, which will then help us to reject false alarms based on additional context not available to the neural bug detector.

6.2 Validation for Neural Bug Detection

We start with a more general view on neural bug detection and we introduce the task of validation for neural bug detection. The following definitions will then help us for the description of our validation approach in Section 6.3

Neural Bug Detectors. Our goal in this chapter is a validation approach that can validate the output of neural bug detectors. As in the previous chapters, we specifically focus on neural bug detectors that are designed to (1) classify a program T as buggy or not, (2) localize the bug location (if any) and (3) propose a repairing patch D. As these three tasks are modeled jointly, we can view a neural bug detector as a function $f : \mathbb{T} \to \mathbb{D}$ that maps any program $T \in \mathbb{T}$ to a patch $f(T) = D \in \mathbb{D}$ which transforms T into a *less buggy* variant T':

$$T \xrightarrow{f(T)} T'$$

Here, the neural bug detector should map T to a patch $D = \emptyset$ if T is already bug-free (and a change is not necessary). We denote $\sigma_D : \mathbb{T} \to \mathbb{T}$ as the patch operation that

applies D to map T to T' (i.e. $\sigma_D(T) = T'$). In practice, neural bug detectors are often trained to detect simple bugs at *function-level* [AJFB21]. Therefore, they might produce *false alarms* and non bug-fixing patches $D = f(T) \neq \emptyset$ because they do not have access to the remaining file context.

Validation for Neural Bug Detection. By relating each raised alarm to a potential patch $D \neq \emptyset$, we find that the problem of validating the output of a neural bug detector becomes a *patch validation problem* [QLAR15]: Given a neural bug detector $f : \mathbb{T} \to \mathbb{D}$, our goal is a patch validator $\mathcal{V} : \mathbb{T} \times \mathbb{D} \to \{0, 1\}$ that accepts bug-fixing patches $D = f(T) \neq \emptyset$ of T and rejects any other patch corresponding to a false alarm. Note that in practice patch validation is commonly addressed via test-based execution [QLAR15]: Given a set of tests, a patch D is considered bug-fixing if T fails at least one test and $\sigma_D(T)$ does not. We however find that a test-based validation is rather limiting in the context of neural bug detection: neural bug detectors are often designed to *statically* analyze the code in scenarios where a complete (test-based) specification might not be available. Therefore, to not restrict the generality of neural bug detectors, we aim to employ large language models trained on code to validate the patches and false alarms generated by a neural bug detector *without* code execution.

Large Language Models. For validating the output of neural bug detectors without code execution, we employ Large Language Models of Code (LLM) [BMR+20]. LLMs model the probability of seeing a given program $T = t_1, t_2, \ldots, t_n$:

$$P_{LM}(T) = \prod_{i=1}^{n} P_{LM}(t_i \mid T_{<i}),$$

where $P_{LM}(t_i \mid T_{<i})$ is the probability of seeing token t_i given the code prefix $T_{<i}$. As the probability tends towards zero for longer sequences, recent work [RHG+16] has used the *entropy* $H_{LM}(T)$ of a given program to judge its probability:

$$H_{LM}(T) = -\log P_{LM}(T).$$

A lower entropy indicates that the LLM assigns a higher probability to a given code snippet. Recent LLMs [FAL+23] support longer file contexts which we aim to exploit in the following. For this, we condition the LLM on an extra context C which we denote by $H_{LM}(T \mid C)$ and $P_{LM}(T \mid C)$.

6.3 An LLM-based Validator for Neural Bug Detection

In the following, we investigate the application of large language models for validating the output of neural bug detectors. For this, we aim to exploit the longer context processing capabilities of recent LLMs to validate patches generated by function-level neural bug detectors. An overview of our approach can be found in Fig. 6.1. Here, the

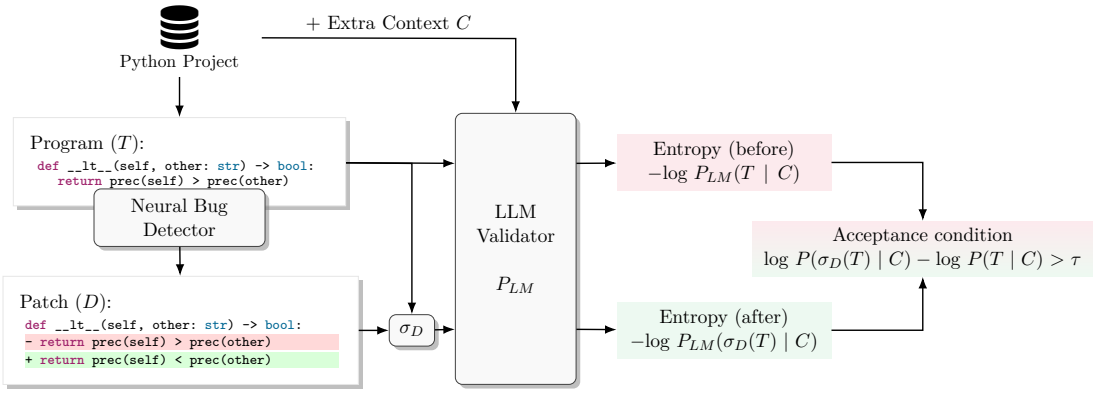


Figure 6.1: Overview of the validation strategy

neural bug detector *wrongly* detects a bug in the given function implementation and proposes a *faulty* patch. In order to identify the false alarm, we employ an LLM to score the code before and after applying the patch. If the LLM assigns a higher probability to the patched code, we accept the patch and otherwise reject it. To achieve a more precise prediction, we experiment with retrieving extra context from the source project. In Section 6.3.1, we describe our LLM-based patch validation strategy in more detail. We employ two strategies to increase the precision of our validation approach: (1) we formulate the task of patch validation as a selective infilling problem in Section 6.3.2 and (2) we retrieve extra context from the code project as described in Section 6.3.3. Finally, we provide further implementation details in Section 6.3.4.

6.3.1 Patch Validation with Large Language Models

To validate patches generated by a neural bug detector, we built upon the observation that (large) language models often assign a higher probability (lower entropy) to the fixed code than to its buggy variants [RHG+16]. To exploit this observation, we employ the following validation strategy: Given an LLM that models the probability $P_{LM}(T)$ of seeing a program T, our validator accepts patches $D \neq \emptyset$ of T only if that increases the probability score of T (as computed by the LLM) significantly:

$$\mathcal{V}_{LM}(T, D) = [\![\log P_{LM}(\sigma_D(T)) - \log P_{LM}(T) > \tau]\!],$$

with $\tau \geq 0$ is an acceptance threshold. Optionally, we can condition on an extra context C denoted by $\mathcal{V}_{LM}(T, D \mid C)$ where we replace the probabilities $P_{LM}(T)$ with the conditioned probabilities $P_{LM}(T \mid C)$. $\mathcal{V}_{LM}(T, D)$ is a special case with an empty extra context.

Relation to Entropy. To further motivate the design of our validation strategy, we relate our strategy to existing research in the (un-)naturalness of buggy code [RHG+16, KBR+20, YKH+24]. For example, Ray et al. [RHG+16] explored how applying a real

bug fix impacts the entropy computed by a language model. They found that the entropy often drops after applying a bug fix, i.e. in many cases:

$$H(\sigma_D(T)) - H(T) < 0,$$

where D is a real bug fix and that the entropy of buggy code is often higher than the entropy of non-buggy code. This finding was later extended to neural language models [KBR+20] and recently to large language models [YKH+24]. Motivated by this finding, we employ the entropy to validate the output of neural bug detectors. In fact, our validation strategy measures the drop in entropy (which is equivalent to the increase in log probability) to accept or reject patches generated by the neural bug detector.

6.3.2 Patch Validation as Selective Code Infilling

Although we can use our validation strategy as described before, we found that the entropy for small code changes, e.g. a single token or statement, over large inputs can become unreliable. Therefore, we propose to exploit the code infilling capabilities of recent LLMs for code [FAL+23].

Infilling Language Modeling. Infilling language models [FAL+23] are trained to model the probability of a code infilling given the surrounding context, i.e.: Assume that we can decompose a program $T = [\texttt{prefix}; \texttt{infill}; \texttt{suffix}]$, then an infilling language model computes the probability:

$$P_{LM}(\texttt{infill} \mid \texttt{prefix}; \texttt{[M]}; \texttt{suffix}),$$

where [M] indicates the position where the code should be infilled[1]. Existing LLMs for code are often trained jointly to model both the probability $P_{LM}(T)$ and the probability of seeing an infill $P_{LM}(\texttt{infill} \mid \texttt{prefix}; \texttt{[M]}; \texttt{suffix})$.

Patch Validation via Selective Infilling. We exploit the fact that neural bug detectors often generate *small* patches $D \neq \emptyset$ that only change a single statement or token. Therefore, we can decompose any program T and its patched variant $\sigma_D(T)$ into

$$T = [\texttt{prefix}; M_\emptyset; \texttt{suffix}] \quad \text{and} \quad \sigma_D(T) = [\texttt{prefix}; M_D; \texttt{suffix}],$$

with $M_\emptyset \neq M_D$. We achieve this by running a simultaneous depth first search on the AST of the code before and after applying the patch D (see Section 4.3.1) until we find the first non-leaf AST node where the two code snippets differ. $M_\emptyset$ and M_D therefore often consists of relatively small expression, e.g. a changed binary operator with its

[1] Note that infilling language modeling is equivalent to masked language modeling. However, as masked language modeling is traditionally used for infilling a single token and infilling is the generalization of this, we stick to the terminology.

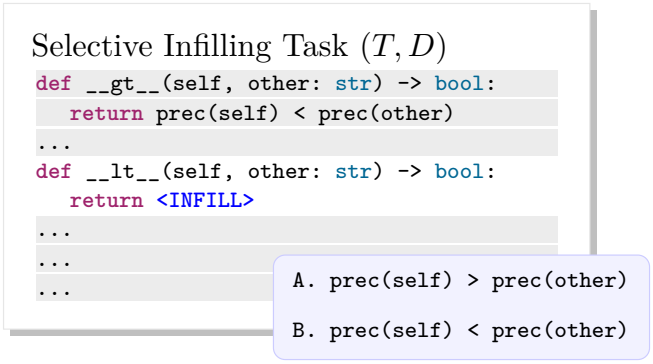


Figure 6.2: A selective infilling problem based on our example program T and the faulty patch D. The LLM has to select between the changed code before (A.) and after (B.) applying the patch. Extra context (gray code) can be provided.

operands. Then, we use the language model to estimate:

$$P_{LM}(T) \approx P_{LM}(M_{\emptyset} \mid T_{[\mathrm{M}]}) \quad \text{and} \quad P_{LM}(\sigma_D(T)) \approx P_{LM}(M_D \mid T_{[\mathrm{M}]}),$$

with $T_{[\mathrm{M}]} = [\texttt{prefix}; [\mathrm{M}]; \texttt{suffix}]$. We find that in this case patch validation becomes a *selective infilling problem* where the language model has to select between the infilling $M_{\emptyset}$ before applying the patch and the infilling M_D after applying the patch D.

Example. We provide an example for a selective infilling task in Fig. 6.2. For this, we employ our transformation to transform the *faulty* patch provided by our neural bug detector into a selective infilling problem. Here, the example consists of a single `__lt__` function where we masked out the comparison between `prec(self)` and `prec(other)`. The language model has to then select which infilling fits the given context better. Note that infilling does not exclude the use of extra context. Hence, the LLM can use the information provided by surrounding context to select the correct infilling A. Because of this, our validator would correctly reject the given faulty patch.

6.3.3 Adjusting the Validator to Different Contexts

Existing neural bug detectors often perform bug detection at function-level [AJFB21]. The key advantage of using LLMs as a *post-hoc* validation strategy is that we can adjust the context provided to the LLM. Therefore, during our experiments, we consider five different types of context: (1) no context, (2) line-level context, (3) function-level context, (4) class-level context and (5) file-level context and their impact on the validation performance. We describe the different context types in the following. An overview of the different context types for our example is given in Fig. 6.3.

No context. As a baseline, we experiment with providing no context at all. Given our decomposition in $[\texttt{prefix}; M_D; \texttt{suffix}]$, we approximate the LM probabilities as

```
def prec(state: str) -> int:
    """Get the precedence index for state.

    Lower index means higher precedence.
    """
    try:
        return PRECEDENCE_LOOKUP[state]
    except KeyError:
        return NONE_PRECEDENCE

# START OF CLASS CONTEXT ----------------------------------------
class state(str):
    """Task state.

    State is a subclass of :class:`str`, implementing comparison
    methods adhering to state precedence rules::

        >>> from celery.states import state, PENDING, SUCCESS

        >>> state(PENDING) < state(SUCCESS)
        True
    ...
    """

    def __gt__(self, other: str) -> bool:
        return prec(self) < prec(other)
    ...
    def __lt__(self, other: str) -> bool:
        return prec(self) > prec(other)
```

Figure 6.3: Example context taken from Celery for our example of a false alarm generated by a neural bug detector. The example is slightly adapted to fit the figure.

follows:

$$P_{LM}(T) = P_{LM}(M_{\emptyset}) \quad \text{and} \quad P_{LM}(\sigma_D(T)) = P_{LM}(M_D)$$

In our example, we would only use the changed part that is marked in the darker red in Fig. 6.3. We include this option to evaluate the impact of the local context (e.g. variable names used in the changed code) without the larger context (e.g. the function signature or documentation).

Line-level context. Ray et al. [RHG+16] originally proposed to compute the entropy of code at a line-level. We therefore estimate the entropy of the changed part conditioned on the rest of the line where the change occurs. For multi-line changes, we employ the line prefix of the start line and the line suffix of the end line of the change as context.

Function-level context. This is the default context provided to a neural bug detector. We consider this setting to evaluate the effectiveness of our validation strategy if it only provided the same context as the neural bug detector. In our example, we hence would include the complete definition of the `__lt__` function, but we omit any other context that appears in the same file.

Class-level context. For functions that occur as a member of a class definition, we consider a class-level context. Here, we include anything that is defined in the class of the changed function. For standalone functions, class-level context is equivalent to

function-level context. In our example, the class-level context (starting in Line 12) would already be sufficient to detect the false alarm.

File-level context. For this context type, we include everything that appears in the same file of the changed function. This includes for example definitions and constants that are used by the considered function. In our example, we hence include the definition of the `prec` function which could also help in identifying the false alarm.

Beyond file-level context. While we could increase the context beyond file-level, we found that most of the considered LLMs are restricted to predictions at file level. Therefore, we leave the integration of further context that goes beyond the context of single file open for future work.

6.3.4 Implementation

To evaluate the impact of using different LLMs with different context types on the validation of patches provided by a neural bug detector, we implemented our validation strategies in a common framework. Our framework mainly builds upon the `transformer` library [WDS+20]. We implemented the inference code to compute the entropy for different LLMs on our examples. For this, we employed three types of LLMs: (1) auto-regressive LLMs, (2) infilling LLMs in PSM mode and (3) infilling LLMs in SPM mode [FAL+23]. Auto-regressive LLMs do not support infilling and hence we compute the entropy based on $P_{LM}(T)$ of the full-sequence. For the infilling models, we found that existing models support one (or both) of two different modes for infilling: (1) prefix-suffix-middle (PSM) mode and (2) suffix-prefix-middle (SPM) mode. In PSM mode, the LLM models the probability:

$$P_{LM}(\texttt{infill} \mid \text{[PRE]}\texttt{prefix}; \text{[SUF]}; \texttt{suffix}; \text{[MID]}),$$

where [PRE], [SUF], [MID] are special tokens to indicate the start of the prefix, suffix and the infilling respectively. The SPM mode uses the suffix first:

$$P_{LM}(\texttt{infill} \mid \text{[SUF]}; \texttt{suffix}; \text{[PRE]}\texttt{prefix}).$$

In this case, the `infill` is a natural continuation of the prefix while the LLM can utilize information from the suffix. We found that for the task of patch validation, the SPM mode often works better. Therefore, if the SPM mode is available, we use the SPM mode, and otherwise we use the PSM mode.

6.4 Evaluation

In our evaluation, we investigate the effectiveness of large language models to validate patches generated by a neural bug detector. Our goal is to show that language models can effectively utilize file-level context to validate the false alarms produced by a neural

bug detector while maintaining its ability to detect and repair real bugs. In the process, the following research question have guided our evaluation:

RQ1 How effective are large language models in validating the output of neural bug detectors?

RQ2 How does the extra file-level context impact the validation performance?

RQ3 How does validation impact the performance of the neural bug detector?

RQ4 Can validation help in finding novel bugs in existing code projects?

For answering the research questions, we collected raised alarms and patches generated by an existing neural bug detector[2]. The neural bug detector is tuned for single token bugs in Python. Therefore, all generated patches address variable misuse, binary operator, unary operator or literal bugs in Python functions.

6.4.1 Evaluation Tasks

To answer our research questions, we evaluate our LLM-based validation strategy on the following two tasks: (1) *real bug fix validation* and (2) *false alarm reduction*.

Real Bug Fix Validation. For evaluating the ability of our validation strategy to detect real bug-fixing patches, we employ the PyPIBugs benchmark [AJFB21]. PyPIBugs consists of 2374 real-world Python bugs and their patches derived from open source projects. The patches are validated via manual inspection and hence each patch likely correspond to a real bug that is fixed by the given patch. After filtering for bug types that our neural bug detector supports, we employ a subset of 2028 real bug fixes that address variable misuses, operator bugs and literal bugs in Python. On this benchmark, the neural bug detector detects 1274 bugs from which it fixes 736 bugs[3] in the same way as the original developer. As we cannot evaluate the correctness of the remaining 538 generated patches, we assume that they are *false repairs*. During our experiments, we evaluate the validator on the task of detecting real bug-fixing patches while rejecting false repairs produced by the neural bug detector (even if this means that we revert to a buggy version of the program).

False Alarm Reduction. To evaluate the ability of our validation strategy to reduce false alarms, we employ the DyPyBench benchmark [BKP24]. DyPyBench collects around 50 open source Python projects with available test suites. We excluded 4 projects where we could not set up the testing process due to dependency problems. As the remaining projects are all highly tested and well-maintained, we assume that they are all bug-free and that any alarm reported by a neural bug detector is a false alarm. For our evaluation, we run the neural bug detector on all considered projects

[2]We employ the best-performing Transformer-based neural bug detector from Chapter 5

[3]We filtered out some bug fixes where the patch produced by the neural bug detector could not be parsed.

and collect all raised alarms and patches. The neural bug detector traversed around 12357 functions and flagged 2505 functions from 33 out of 46 projects as potentially buggy (a false alarm rate of 20.3%). We run the available tests on the code before and after applying the generated patches to confirm the false alarms. We found that 1788 of the 2505 raised alarms (71.4%) can be confirmed by the tests to be false alarms. The remaining 28.6% of code changes are not detected during test execution and hence remain unconfirmed. For our experiments, we consider both confirmed and unconfirmed false alarms.

Additional context. Key to our validation strategy is that we have access to additional context provided in the original code repositories. For this, we retrieved the file context for each function in our benchmarks. For DyPyBench, we simply gathered the file context from the file where the original function appears. For PyPIBugs, we had to crawl the commit history of the original repositories. We collected the version of the modified files before applying the respective real bug fix. During this process, we found that the file context for 65 out of 736 detected and fixed real bugs and for 46 out of 538 false repairs a file context is no longer available. In these cases, we use the function-level context as the maximal context used for validation. In total, we mined over 900 repositories to obtain the original file contexts.

Metrics. To evaluate the effectiveness of our validation strategy on the real bug fix validation task, we report *precision* and *recall*. For this, we consider patches generated by the neural bug detector that reproduce real bug fixes as *correct patches*. Further, the patches generated by the neural bug detector (either correct or incorrect) can be accepted or rejected by the validator. Based on this definition, we can define precision and recall in this context as follows:

$$\text{precision} = \frac{\#\text{correct accepted patches}}{\#\text{accepted patches}} \quad \text{recall} = \frac{\#\text{correct accepted patches}}{\#\text{correct patches}}$$

For the task of false alarm reduction, we report the *false positive rate* (FPR) which is in this context the percentage of false alarms accepted by the validator. We report the FPR for both confirmed and unconfirmed false alarms.

6.4.2 Baselines

We employ several baselines for evaluating the effectiveness of our LLM-based validation strategy.

Thresholding. As a baseline, we consider *thresholding*, which was recently suggested by Dinh et al. [DZT+23] for neural bug detection. For this, we exploit the fact that a neural bug detector f often models the probability $P_f(D \mid T)$ that D fixes a bug in T. The main idea of thresholding is then to only accept $D \neq \emptyset$ where $P_f(D \mid T)$ exceeds a certain threshold (i.e. $P_f(D \mid T) > \rho$).

Test-Based Patch Validation. A test-based patch validation strategy can be an effective way to reduce the number of patches related to false alarms (if user-specified tests are available). For example, assume that we have access to a test suite and all tests pass for the existing code base. If we assume that the test suite represents a form of ground truth, we can reject all raised alarms and patches that lead to failing tests. As we have seen before, this would automatically lead to a substantial reduction in false alarms generated by the neural bug detector. We compare our LLM-based validation strategy with a test-based patch validation strategy to better understand the similarities and differences between those two approaches.

Large Language Models. During our experiments, we employ eight different LLMs. As a baseline LLM that supports file-level code infilling in Python, we employ InCoder-1B and InCoder-6B [FAL+23] with one and six billion parameters respectively. To evaluate the effectiveness of our infilling-based strategy, we also evaluate against CodeGen-350M and CodeGen-2B (mono) [NPH+23] which are 350M and 2B parameter models that are trained for *auto-regressive* code completion in Python. As they do not support code infilling, we compute the entropy of the full code sequence. We also consider Qwen2.5-Coder [HYC+24]. Qwen2.5-Coder is a more recent large language model trained on a variety of programming languages that also supports file-level code infilling. At the same time of writing, Qwen2.5-Coder comes in two variants: a 1.5B variant and a 7B variant. To further evaluate the impact of scale, we consider CodeLlama-7B and CodeLlama-13B [RGG+23], a 7B and 13B variant trained on the popular Llama 2 architecture [TMS+23].

6.5 Results

In the following, we present our experimental results to answer our research questions.

6.5.1 RQ1 - Validating Bug Fixes and False Alarms

To answer **RQ1**, we evaluate our validation strategy on patches and false alarms generated by our neural bug detector.

Experimental setup. We evaluate our LLM-based validation strategy against our baselines on the tasks of real bug fix validation and false alarm reduction. We group LLMs according to their size: LLMs with fewer than 2B parameters are considered small and large otherwise. For the LLM-based validation strategies, we default to an acceptance threshold $\tau = 0$. For the comparison with thresholding, we plot the performance of the LLM-based strategies as a ROC curve that measures the trade-off between FPR and recall of the different strategies at different threshold. To measure the effectiveness, we also report the area under the curve (AUC) of the different strategies. A higher AUC indicates a better trade-off between FPR and recall. All evaluated LLMs have access to the full file context.

Table 6.2: Comparison between different LLMs as Validators at $\tau = 0$

Validator	Real Bug Fixes		False Alarms	
			Confirmed	Unconfirmed
	Precision↑	Recall↑	FPR↓	FPR↓
Small				
CodeGen-350M	73.3	72.4	18.8	18.8
InCoder-1B	69.2	78.9	21.5	35.3
Qwen2.5-Coder-1.5B	**76.2**	**81.8**	17.8	27.9
CodeGen-2B	75.7	70.5	**13.1**	**12.6**
Large				
InCoder-6B	73.5	80.0	17.0	21.1
Qwen2.5-Coder-7B	77.8	82.3	16.1	22.7
CodeLlama-7B	**79.6**	**89.0**	**8.9**	**20.6**
CodeLlama-13B	**80.9**	**89.0**	**8.1**	**17.3**

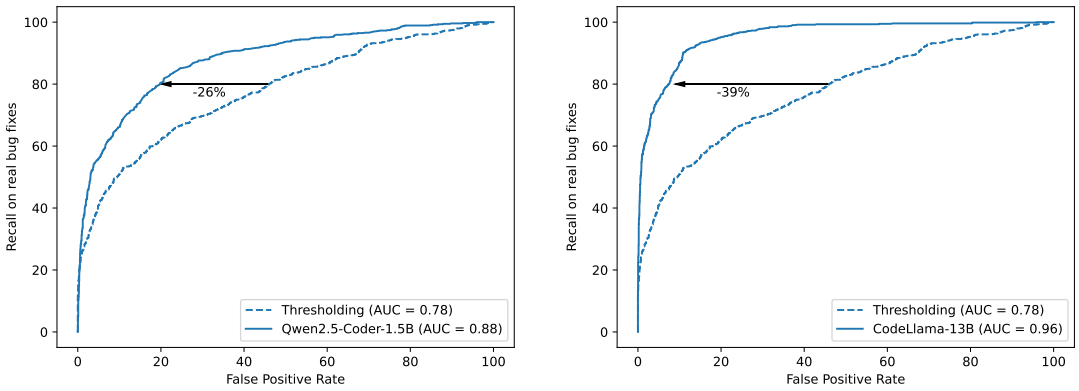
Results. The results of our comparison between LLM-based validation strategies are shown in Table 6.2. We report precision and recall on the real bug fix validation task and FPR on the false alarm reduction task in percent.

Small LLM-based validators are already effective. We find that already small LLM-based validators (up to 2B parameters) are effective in reducing the number of patches related to false alarms. They reduce the number of confirmed false alarms by up to 86.9% (to an FPR of 13.1% for CodeGen-2B) and the number of unconfirmed false alarms by up to 87.4%. The most effective small LLM for real bug fix validation is Qwen2.5-Coder-1.5B with a recall of 81.8% and a precision of 76.2%.

Detecting unconfirmed false alarms is significantly harder. We find that across all LLM-based patch validators (except for the CodeGen models) detecting unconfirmed false alarms is significantly harder. The FPR increases by up to 13.8% for small LLMs and by up to 11.7% for larger LLMs. This could indicate that unconfirmed false alarms are harder to detect or that the set of unconfirmed false alarms in fact include real bug fixes. We discuss this further in Section 6.6.

Impact of Scale. We find that the scale of LLMs is the most important factor for false alarm reduction. In a direct comparison between InCoder-1B and InCoder-6B and Qwen2.5-Coder-1.5B and Qwen2.5-Coder-7B, we find that the scale difference has negligible impact on recall. At the same, precision and FPR improves significantly (with a precision improvement of up to 4.3% and an FPR reduction of up to 14.2% on unconfirmed false alarms). The same holds true if we compare CodeLlama-7B and CodeLlama-13B. Here, the recall does not change while the FPR reduces by up 3.3% (on unconfirmed false alarms). In total, the most effective large LLM is CodeLlama-13B.

Impact of LLM Choice. We find that even at the same size the difference between LLMs



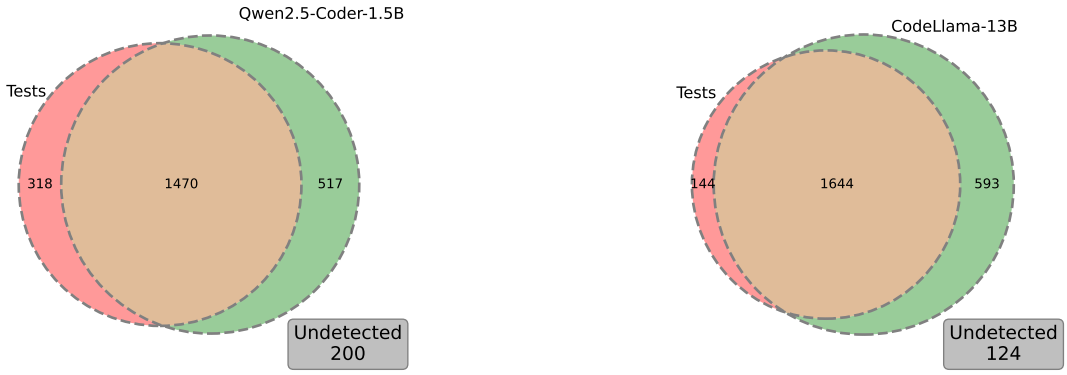
(a) Small Validator (b) Large Validator

Figure 6.4: Comparison with Thresholding

(e.g. Qwen2.5-Coder-7B and CodeLlama-7B) can be significant. Although they have approximately the same number of parameters, CodeLlama-7B is significantly more effective in detecting real bug fixes and in reducing the number of false alarms.

Comparison with Thresholding. We compare the best-performing small LLM-based validation strategy (Qwen2.5-Coder-1.5B) and the best-performing large LLM-based validation strategy (CodeLlama-13B) against a simple thresholding approach. Our results are shown in Fig. 6.7. We plot the false positive rate (on all false alarms) against the recall on real bug fixes for both thresholding and the validator at various thresholds. We report area under curve (AUC) and the reduction in false alarms at a recall of 80% (approximately the recall achievable with $\tau = 0$). We find that applying a validator leads to a better trade-off between recall and FPR. Already the small LLM-based validator reduces the number of false alarms by 26% by maintaining the same recall. Employing a larger validator can reduce the number of false alarms further by another 13% at a recall of 80%. We note that we can a significantly higher improvement at higher recall level by up to 33% at a recall of 90% for the small validator and by up to 59% at a recall of 95% for the larger validator.

Comparison with Test-based Patch Validation. We are interested how the LLM-based patch validation compares to a test-based patch validation. While we already have seen that the LLM-based patch validation can detect unconfirmed false alarms, we aim to provide concrete numbers for the comparison. Therefore, in Fig. 6.5, we report the number false alarms that both tests and LLM discovered and the alarms that only one of the technique discovered in a Venn diagram. Additionally, we report the number of false alarms that no validation technique detected. We find that for both small and large LLM-based validator, there is a significant overlap of false alarms that both tests and LLM detected. The LLM-based strategy is able to detect between 517 and 593 more false alarms generated by the neural bug detector than the test-based strategy. However, there are also between 144 and 318 false alarms that are only detected via



(a) Small Validator (b) Large Validator

Figure 6.5: Comparison to Test-based Patch Validation at $\tau = 0$

tests. Between 124 and 200 false alarms remain undetected even if we combine testing and LLM-based patch validation.

Based on these observations, we conclude for **RQ1**:

> LLM-based patch validation strategies are effective in validating the output of neural bug detectors. In comparison to thresholding, employing an LLM-based patch validator can significantly reduce the number of false alarms at the same recall level. In addition, it can detect false alarms missed by a test-based patch validation.

6.5.2 RQ2 - Impact of Context

We hypothesized that a crucial part of our validation strategy is the ability to incorporate extra context not available to the neural bug detector. Therefore, to evaluate our hypothesis and at the same time answer **RQ2**, we evaluate the impact of the context on the different validation strategies.

Experimental setup. We evaluate the best performing small and large LLM-based validation strategy on the task of real bug fix validation and false alarm reduction. We focus here on the ability to detect real bug fixes (recall) and the overall false positive rate. During our experiments, we evaluate the impact of restricting the available context. Therefore, we restrict the context to file-level, class-level, function-level and line-level context. In addition, we also experiment with *no context* where the validator has to decide solely based on the changed code.

Results. Our results are shown in Table 6.3. We report both recall on real bug fixes and the FPR across all false alarms for each context type.

Additional context can help for rejecting false alarms. We find that increasing the available context can help in decreasing the FPR (with the exception for the small validator when switching to a file-level context). For CodeLlama-13B, we find that the FPR significantly decreases by up to 4.8% when switching from a function-level to

Table 6.3: Impact of Different Context Types on Validator at $\tau = 0$

Context	Validator			
	Small Qwen2.5-Coder-1.5B		**Large** CodeLlama-13B	
	Recall↑	FPR↓	Recall↑	FPR↓
No Context	43.2	36.0	48.6	32.7
Line Context	57.3	32.3	62.9	30.9
Function Context	**88.5**	21.5	**90.6**	15.5
Class Context	85.6	**18.2**	89.5	11.9
File Context	81.8	20.7	89.0	**10.7**

a file-level context. The FPR of Qwen2.5-Coder-1.5B improves by 3.3% when switching to the larger class-level context. There is still a small decrease of 0.8% for Qwen2.5-Coder-1.5B when switching from a function-level context to a file-level context.

Increasing the available context beyond function-level does not improve recall. We find that increasing the context beyond function-level can have a negligible (in the case of CodeLlama-13B) or negative impact (in the case of Qwen2.5-Coder-1.5B) on the recall performance. This could indicate (1) that larger LLMs better utilize the additional context and (2) that the function-level context is in most cases sufficient to confirm the bug detected at function-level. We believe that a decrease in recall is acceptable in favor of a significantly improved FPR.

Context is important. We find that decreasing the context below function-level significantly decreases the performance. At a recall close to 50%, the LLM-based validation strategy becomes worse or equivalent to a random strategy (that tosses a coin to accept real bug fixes). The validation is still relatively effective in rejecting patches related to false alarms. This could indicate that patches related to false alarms are often easy to reject while to accept real bug fixes a larger context is necessary.

Overall, we conclude for **RQ2**:

> Incorporating extra file-level context which is often not available to the neural bug detector can significantly boost the ability of the LLM based validator to reject false alarms. For the smaller validator, incorporating larger contexts (beyond function-level) can result into a trade-off between recall and FPR.

6.5.3 RQ3 - Impact on Neural Bug Detection

For answering the previous research questions, we evaluated the effectiveness of the LLM-based validator to validate the output of a neural bug detector. Now, for answering **RQ3**, we are interested how employing a validator impacts the overall performance of the neural bug detector.

Table 6.4: Evaluation results for the validated neural bug detectors on our benchmark tasks.

	All		**VM**		**Ops**		**Literal**	
	Joint.↑	FAR↓	Joint.↑	FAR↓	Joint.↑	FAR↓	Joint.↑	FAR↓
Neural Bug Detector	36.3	20.3	40.1	3.3	30.8	12.5	26.7	4.5
+ Thresholding	21.2	3.3	28.7	**0.5**	9.2	2.3	10.5	**0.5**
+ Small Validator	29.6	4.2	35.0	0.7	20.7	2.3	**24.4**	1.1
+ Larger Validator	**32.2**	**2.2**	**38.2**	**0.5**	**23.1**	**1.1**	22.1	0.6

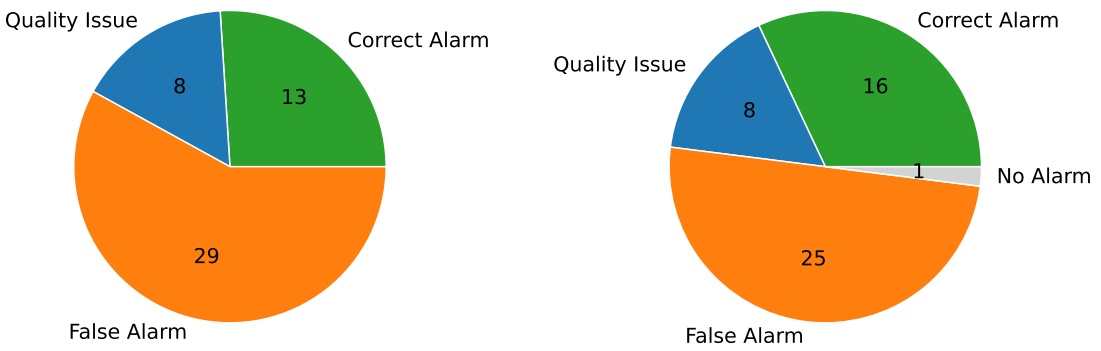
Experimental setup. We evaluate the impact of the best-performing small validator and best-performing large validator on the performance of the neural bug detector. For this, we evaluate the neural bug detector on the 2028 real bugs from PyPIBugs and the 12357 likely bug-free functions from DyPyBench. We measure the performance of the neural bug detector before and after applying our validation strategy. As a baseline, we employ thresholding with a threshold of 0.9. For the validator, we keep the acceptance threshold of $\tau = 0$. The LLM-based validators have access to the file-level context.

Results. Our experimental results are shown in Table 6.4. We report the *joint* recall (Joint.) of detecting and fixing a real bug before and after validation on the overall benchmark (All) and for variable misuse (VM), operator (Ops) and literal bugs (Literal). In addition, we also report the *false alarm rate* (FAR) over the complete benchmark and per bug type. The FAR measures the ratio of false alarms over the total number of likely bug-free functions. For the evaluation per bug type, we group the false alarms according to the bug type the corresponding patch aims to fix.

Validation can reduce the false alarm rate by up to a factor of 11x. We find that by applying a validator to the output of a neural bug detector helps us to significantly reduce the false alarm rate. Over all benchmark tasks, applying the large validator can reduce the false alarm rate by a factor up to 9x. For the individual bug types, we find that bigger gains can be achieved (by a factor of up to 11x for operator bugs). The small validator achieves a reduction of 4.8x for a bug types and a reduction of up to 5.4x on the individual bug types.

Validation hurts joint recall slightly. As expected, applying a validator decreases the number of real bugs found by the neural bug detector slightly. The joint recall drops by 1.9% (VM) up to 7.7% (Ops) when employing the large validator. For the small validator, the joint recall drops by 6.7% on all bug types and by up to 10.1% on the individual bug types.

Comparison to Thresholding. In comparison to thresholding, we find that our validation strategies can significantly improve the joint recall while maintaining a similar or lower false alarm rate. Across all bug types, the large validator can improve the joint recall by up to 13.9% while maintaining or even reducing the FAR (e.g. by 1.1% for the All category and 1.2% for the Ops category). The small validator also boosts the joint



(a) Without Validation

(b) With Validation

Figure 6.6: Impact of validation on the ability of the neural bug detector to detect novel bugs in Python projects.

recall significantly (by up to 11.5% on Ops). However, employing the small validator might not always result in a decrease in FAR.

Finally, to conclude for **RQ3**, we find that:

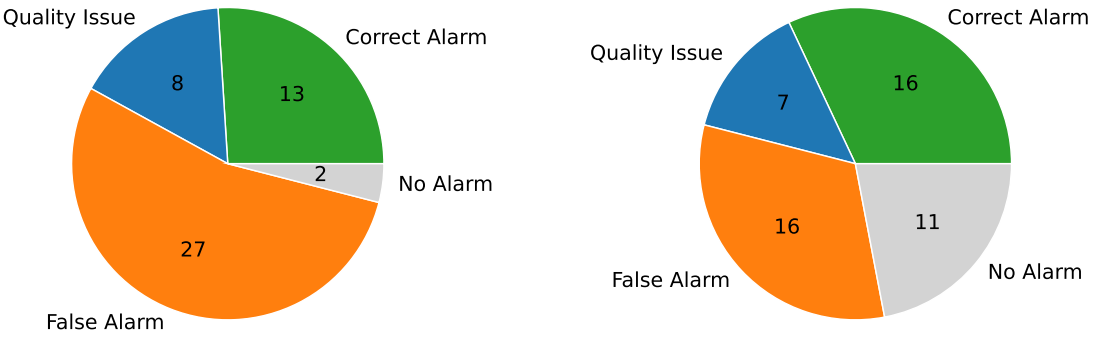
> External validation with LLMs can significantly reduce the false alarm rate of neural bug detectors while maintaining their performance on real bugs. In comparison to thresholding, external validation achieves a better trade-off between joint bug detection and repair recall and false alarm rate.

6.5.4 RQ4 - Finding Novel Bugs in Public Projects

To get a better understanding of how the combination of neural bug detector and validator works in practice and to answer **RQ4**, we evaluate the combination on the task of scanning recent open source projects for real bugs.

Experimental setup. We evaluate the impact of validation on the ability of the neural bug detector to detect novel bugs in *recent* open source projects. For this, we employ the neural bug detector to scan the 50 most popular Github projects that were created after September 2022[4]. An overview of all projects that we scanned is provided in Appendix A.3.1. For each project, we sort the raised alarms by the probability that the code contains a real bug (as computed by the neural bug detector). We then manually inspect the top 5 alarms (before and after validation) per project and categorize them into (1) *Bugs* that cause the wrong program behavior, (2) *Quality Issues* that are not bugs but impair code quality (e.g. unused variables or code that does not conform to code conventions), and (3) *False Alarms*.

[4]Both the training data for the neural bug detector and the validator consists of data mined before September 2022. We used the Github API to retrieve the names of the top 50 Python repositories sorted by the number of stars that were created after September 2022.



(a) Thresholding

(b) Thresholding + Validation

Figure 6.7: Impact of post-hoc validation strategies on the ability of the neural bug detector to detect novel bugs in Python projects.

Results. Our results are shown in Fig. 6.6. We report the number of projects with at least one correctly reported bug (Correct Alarm). The remaining projects are categorized in projects with quality issues (Quality Issue), false alarms (False Alarm) and projects where no alarm is raised (No Alarm). An overview of all bugs and quality issues that we found in the process of our manual review is given in Appendix A.3.2. Overall, we find that:

The neural bug detector alone reports (false) alarms on all projects. We find that the neural bug detector alone produces an excessive amount of alarms. In total, across all projects, the neural bug detector reports more than 17K alarms which corresponds to 354 alarms per project on average. The sheer number of raised alarms might overwhelm a software developer. Reviewing a subset of the raised alarms can however still help to find novel bugs. For example, by reviewing the top-5 alarms we find a novel bug in 13 projects and a quality issue in 8 projects.

Validation can help focusing the reviewing efforts. By employing our LLM-based validator, the number of raised alarms reduces significantly: After validation, the combination of neural bug detector and validator only reports around 1.7K alarms (10% of all alarms) across all projects which reduces the number of alarms per project to 35 alarms on average. Interestingly, we find that validation helps us to focus our reviewing efforts. The validator filters a high number of false alarms that are assigned a high confidence by the neural bug detector. When we remove these alarms with the help of validation, we focus our reviewing efforts on alarms that more likely indicate a real bug. In fact, by reviewing the top-5 alarms per project after validation, we were able to find novel bugs in three new projects.

Impact of Thresholding. We additionally experiment with applying thresholding (with a threshold of 0.9) to filter out alarms for which the neural bug detector is less certain. Our results for applying thresholding before and after validation is shown in Fig. 6.7. By

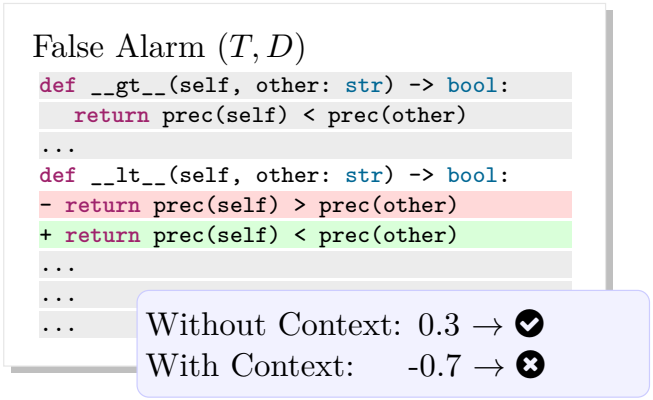


Figure 6.8: Impact of file-level context on our example false alarm. The validator correctly rejects (with a score of -0.7) the false alarm after the file-level context is available.

employing a threshold, we find that the number of alarms already reduces significantly – from 17K alarms to 2K alarms (reducing the average number of alarms per project from 354 to 45). Combining thresholding and validation reduces the number of alarms further to 304 alarms which averages to around 6 alarms per project. Interestingly, we find that most alarms that we confirm to be correct alarms are assigned a high confidence by the neural bug detector. As a consequence, we find that even after applying thresholding and validation a bug can still be discovered in all 16 projects (as shown in Fig. 6.7).

We conclude for **RQ4**:

> Validation can help in finding novel in recent code projects. By filtering out false alarms that are assigned a high confidence by the neural bug detector, the developer can focus more on alarms that help find and fix real bugs. Combing thresholding and validation helps to reduce the number of alarms significantly.

6.6 Discussion

In this section, we now take a more qualitative look at our experimental results. In particular, we are interested in the cases where the LLM-based validator works and where it fails.

File context helps in rejecting false alarms. Our experimental results suggest that the additional file context can help in rejecting false alarms. Therefore, we again consider our initial example of a false alarm that cannot be detected without extra context. The example is shown again in Fig. 6.8. We find that in fact the extra file-level context helps in rejecting the patch and the corresponding false alarm. Before the (large) validator has access to the file-level context, it accepts the patch with a score of 0.3 (for $\tau = 0$). After having access to the remaining context (e.g. the code in the

Table 6.5: Example of Potential Software Bug found in DyPyBench [BKP24]. Code is reformatted to fit the figure.

Example	Description

Potential bug in Zulko/MoviePy
`- quantizer = 0 if opt != 0 else "nq"`
`+ quantizer = 0 if opt == 0 else "nq"`
`writer = imageio.save(`
`  filename, duration=1.0 / fps, quantizer=`
`  → quantizer,`
`  → palettesize=colors,`
`  → loop=loop`
`)`
	`imageio` has two options for quantization `'wu'` and `'nq'` for image quantization (also selectable via `0` and `1`). By using the `!=`, the quantizer `'nq'` is not selectable. This bug is detected by our neural bug detector, but missed by the existing tests.
	Fix: Replace `!=` with `==`.

Table 6.6: Example of Confirmed Software Bug found in `run-llama/llama_index`. Code is reformatted to fit the figure.

Example	Description

Confirmed bug in run-llama/llama_index
`  client = None`
`  if provided_client is not None:`
`-  client = client`
`+  client = provided_client`
	The implementation checks if a client is provided but does not use the provided client. The bug causes a `NoneType` exception when a client is provided. See Bug Report (Issue #16173).
	Fix: Replace `client` with `provided_client`.

gray part), the validator correctly identifies the false alarm and rejects the given patch with a score of -0.7. This showcases the importance of file-context for validating false alarms.

Not all False Alarms are False. We reviewed the 124 raised alarms and patches that are not detected by the test suite and that are accepted by our large validator. As the projects in DyPyBench are all highly popular projects that are extensively tested both via test suites and in practice, we assumed in our evaluation that all raised alarms on these projects correspond to false alarms. Therefore, we were quite surprised to find some raised alarms actually relate to potential software bugs in critical parts of the software. An example is shown in Table 6.5. Here, the neural bug detector detected a potential bug in Zulko/MoviePy which was later confirmed by our validator. MoviePy is a popular video editing package for Python, and it provides utilities to process and save video files. The example shows a potential bug in the `write_gif_with_image_io` function. The user of the function can choose between two quantization options for encoding the video as a gif. However, due to the way the if-condition is constructed it is not possible to select the second quantization method. This behavior is likely unintended and – as testing for quantization methods used in the image encoding is difficult – it is also missed by the test suite.

Novel Bugs in Open Source Projects. During our manual inspection in Section 6.5.4, we found in total 20 bugs across 16 projects that are confirmed by our validator. A complete list of all found bugs is given in Appendix A.3.2. While most of the found bugs are completely novel, we found that one bug was discovered shortly after we scanned the project. The bug and its fix is shown in Table 6.6. Here, the neural bug detector detected a potential bug in run-llama/llama_index. LlamaIndex

Table 6.7: Example of Unconfirmed False Alarm found in DyPyBench [BKP24]. Code is reformatted to fit the figure.

Example	Description

Unconfirmed False Alarm in mwaskom/seaborn
`def __call__(self, data, var):`
`  """Aggregate over ‘var‘ column ..."""`
`  vals = data[var]`
`  ...`
`  if self.error_method is None:`
`  err_min = err_max = np.nan`
`- elif len(data) <= 1:`
`+ elif len(vals) <= 1:`
`  err_min = err_max = np.nan`
`  ...`
	The function aggregates over all values in the column `var` to estimate an error bound. For a table with one or zero entries, an error bound cannot be computed. Typically, one would check the number of elements in the column which is detected by the neural bug detector and confirmed by the validator. However, in this specific case, `len(data) == len(vals)` and hence using `data` is not a functional bug.

is a popular framework for augmenting LLMs with additional knowledge stores, and it provides utilities to connect to existing databases. The example shows a bug in the `create_neptune_database_client` function. The user can provide a client implementation when connecting to a Neptune database. However, the implementation ignores the provided client which causes a `NoneType` exception. Surprisingly, even though the developers of the project employed a set of tests to validate the code, the bug was first discovered by a user of LlamaIndex and reported in Issue #16173. Therefore, the neural bug detector (in combination with our validator) could have been used to detect and fix the bug earlier, before it became a part of the production code.

Failure cases. While our combination of neural bug detector and validator allowed us to detect some real bugs, our validator still accepts some false alarms that are real false alarms. For example on DyPyBench, we found that most of them that are missed by the test suite as well as by the validator are related to semantics-preserving changes. An example in the mwaskom/seaborn project is shown in Table 6.7. While the proposed patch improves the implementation by making size check more explicit, it does not fix a bug. The change is not detected by the test-suite as the change is semantics-preserving. Our LLM-based validator likely accepts the change as using the variable `vals` instead of `data` is more natural in this context. The remaining false alarms that are accepted by the validator often correspond to uncommon coding styles. An example for this is shown in Table 6.8. The neural bug detector detects a bug which is later confirmed by the validator. However, as the validator only processes the file context, it is not aware that code change leads to a type error. This false alarm is rejected by executing the test suite on the changed code. Ultimately, we find in our experiments that both testing and file-level LLM-based patch validation complement each other well. Therefore, we believe that the most effective strategy for validating the output of neural bug detectors is likely a strategy that both utilizes an execution-based and LLM-based patch validation.

Table 6.8: Example of a Confirmed False Alarm that is still accepted by the validators found in DyPyBench [BKP24]. Code is reformatted to fit the figure.

Example	Description

Confirmed False Alarm in PyFilesystem/pyfilesystem2
`# validate individual paths`
`_src_path = self.validatepath(src_path)`
`_dst_path = self.validatepath(dst_path)`
`...`
`# check dst_path does not exist if we are not → overwriting`
`- if not overwrite and self.exists(_dst_path):`
`+ if not overwrite and self.exists(dst_path):`
`    raise errors.DestinationExists(dst_path)`
	The related function checks whether it is valid to copy a file from `src_path` to `dst_path`. `src_path` and `dst_path` are both path objects. `_src_path` and `_dst_path` are normalized string representations of the two paths. As indicated by the comments, the intention is to check if `dst_path` exists and throw an error if it exists. However, the code checks for the normalized path `_dst_path` which is not equal to `dst_path`. Therefore, the neural bug detector *wrongly* reports an error and the fix is accepted by the validator. The test suite detects the change as `exists` (defined in the parent class in a different file) expects a string as the inputs and therefore crashes if provided with a path object.

6.7 Threats to Validity

We discuss potential threats to validity of our experiments. As a part of the discussion, we distinguish between external, internal and construct validity.

External validity. In our experiments, we focused on raised alarms and patches generated by a neural bug detector for single token bugs in Python. Although our method is general enough to be applied to any type of neural bug detector (that generates patches as part of the detection process), our experimental results may not generalize to other neural bug detectors which support different types of bugs or programming languages.

Other Bug Types. The entropy computed by language models is often significantly impacted by the size of the program. Existing strategies [ZYH+23] to circumvent this problem often over- or under-correct. For validating single token patches generated by neural bug detectors, this is typically not a problem as the size of the patched program does not change significantly. However, our findings might not generalize to bug detectors that generate larger patches. One way to address this in the future is by training a sequence classifier that computes the score (nearly) independently of the size of the program.

Other Programming Languages. The choice of programming languages might significantly impact the effectiveness of our validation approach. In our experiments, we focused on Python programs. Although existing LLMs are often trained on a variety of programming languages and their performance often generalize effectively to different languages, the overall effectiveness of our validation method might vary between programming languages.

Selection Bias. Another threat to validity might be our choice of LLMs. We chose representative LLMs at a range of 350M to 13B parameters. However, there exists

much larger LLMs of code that might have different performance statistics. We decided to limit ourselves to LLMs of these sizes because we are limited by our hardware. It is more likely that regular users of neural bug detectors have similar constraints. Large LLMs might still be more effective in detecting faulty patches and reducing the number of false alarms.

Memorization. The precise details of the training data used for training LLMs is often not available. As existing LLMs are trained on large parts of existing Git repositories, it is likely that there is at least a small overlap with the benchmark tasks considered in our experiments. Our experimental results still suggest that our validation approach can help to reject new false alarms generated by a neural bug detector (which cannot be part of the training set) and with the help of the validator we were able to identify novel bugs. However, our results might still not generalize to truly novel code or code that does not appear in public repositories.

Internal validity. We discuss potential threats to the internal validity of our experiments.

Labeling bias. During our experiments, we assumed the patches that do not replicate real bug fixes or that are generated for projects in DyPyBench are incorrect. However, as we already discussed in Section 6.6, we found that not all false alarms are false and some fix real bugs. In addition, our examples of real buggy code might contain multiple types of real bugs and hence the patches that are different from the human provided patches might still be bug-fixing. Therefore, our experimental results might only represent a lower bound on the performance of the respective techniques.

Implementation bias. As a part of our evaluation, we developed several tools to automate the evaluation process. Even though we built upon highly tested libraries for LLM inference, our implementation is still a research prototype and might contain bugs. In addition, the LLMs are often highly sensitive to the way they are prompted. Therefore, even small mistakes in the implementation of our validation strategies could impact their performance significantly. For this reason, we manually inspected the generated inputs to the LLM and confirmed that they are similar to those generated by the reference implementation provided by the original authors.

Construct validity. A key assumption of our work is that (large) language models assign a higher probability to fixed code than to buggy variants of the code. We implicitly assume that there is a causal relationship between the probability computed by an LLM and the buggyness of the code. Hence, we reject all patches that do not increase the probability. Although our experiments show that this validation strategy is highly effective for rejecting false alarms, recent work [JLZ+22] suggests that real bugs alone are not the cause for a lower probability (higher entropy). Still, we find that using entropy can be used as an effective heuristic for reducing false alarms in neural bug detection.

6.8 Related Work

In this chapter, we have investigated the application of large language models for validating the output of neural bug detectors. Our main goal was to reduce the number of false alarms produced by a neural bug detector by integrating further context. In the following, we discuss the most closely related work that address the problem of false alarms in static and neural bug detection. Furthermore, we also compare with techniques used in automatic program repair that address the correctness of patches.

False Alarms in Bug Detection. False alarms in (static) bug detection is a well-known and well-studied problem [AP10, SFZ11, JSMB13, SAE+18, BBC+10, CDD+15]. For example, Johnson et al. [JSMB13] studied why developers do not use a static bug detector like FindBugs [HP04]. They found that false alarms is one of the key reasons why users dislike using them. A more recent study at Google [SAE+18] confirmed that Google engineers easily loose trust in the tool output if most of the reported alarms are false alarms. Therefore, false alarms in static bug detectors is a serious risk that might limit the usability of existing bug detection tools. Because of this, previous work have tried to address the problem of false alarms with techniques to rank and validate alarms raised by static bug detectors [SFZ11, YJJ14, KMJ+22b, KAL22, YWG+24]. However, as most static bug detectors only raise alarms but do provide a patch[5], they had to rely on features of the raised alarm, e.g. the type of warning, and the characteristics of the code for which the alarm is raised. In contrast, we exploit the fact that neural bug detectors often relate alarms to patches. This has allowed us to reject false alarms effectively based on the provided patch. False alarms are also a well-known problem in neural bug detection [VKM+19, AJFB21, HBV22]. Allamanis et al. [AJFB21] found that only a small fraction of the reported alarms of a neural bug detector are related to real bugs. He et al. [HBV22] suggested that the high false alarm rate is due to distribution shift between training and testing. By training on a more realistic distribution, they achieved a significantly higher precision at the cost of a lower recall. We hypothesized in this chapter that this trade-off happens because existing neural bug detectors are limited to a function-level context. Therefore, we proposed a post-hoc validation approach based on large language models which allowed us to significantly reduce the number of false alarms while maintaining the ability of the neural bug detectors to detect and fix bugs *with* the help of extra file-level context.

False Positives in Automatic Program Repair. False positives, in the form of patches that are not bug-fixing, are also a problem in automatic program repair (APR) [SBLB15]. Existing APR tools [TWB+19b, CKT+21, LWN20, LPP+20b, BSPC19b, CDAR20, GPKS17, LAR17, TPW+19, YMM22, ZSX+21, JLT21b] often employ test-based execution to first detect and localize a bug and then to validate the generated patches.

[5]Some static bug detectors offer ‘quick fixes‘ [HP04]. However, as they only provide fixes for some bugs and not all static bug detectors propose them, most techniques have to rely on the tool output.

However, a key problem in APR is the fact that this process can result in *over-fitting* patches or false positives. These over-fitting patches are patches that pass the test suite but do not fix the bug. A common solution to address this problem is *automated patch correctness assessment* (APCA) [XLZ+18, YZLT17b, TLK+20, YGM+22, TTH+22, LWWM22]. APCA ranks or validates patches generated by an APR tool according to its likelihood to be a real bug-fixing patch. Most APCA techniques utilize the fact that they are part of or are applied subsequently to a test-based patch validation. For example, PATCH-SIM [XLZ+18] executes the test suite both on the code before and after applying the potential patch and compares the execution traces. Opad [YZLT17b] employed a fuzz tester to generate further test cases. There also exists APCA methods that use static features of code [TLK+20, YGM+22, TTH+22, LWWM22]. PatchZero [ZXK+23, ZXK+24] is one of the most recent method that uses large language models provided with examples of over-fitting patches and test execution results to detect new over-fitting patches. While some existing APCA methods are in principle applicable for validating patches in the context of neural bug detection, our experimental results show that there are unique challenges. For example, APCA methods are seldom confronted with semantics-preserving code changes as a semantic change is prerequisite for passing test-based patch validation. In addition, most methods assume the existence of a test suite which is often not available in the context of neural bug detection. We still believe that the patches generated by a neural bug detector could be an interesting target for evaluating existing APCA methods. We leave this open for future work.

Finally, concurrent to us, Yang et al. [YKH+24] explored the application of entropy of LLMs in context of APR. They also employed the entropy of LLMs for APCA and found the difference in entropy can be an effective metric for identifying over-fitting patches. We confirmed in this work that the entropy of large language model can be effectively used for validating patches generated by neural bug detectors. In addition, we compared our validation method against a test-based patch validation and evaluated the impact of different context types on the validation performance. Therefore, this chapter can be seen as an extension of the work of Yang et al. [YKH+24] to the setting of neural bug detection.

6.9 Contributions and Conclusions

In this chapter, we explored the application of large language models for validating the output of neural bug detectors. Our key idea was to formulate the task of validating the output of neural bug detectors as a patch validation problem. As a result, we could use large language models to estimate the probability of seeing a program before and after applying a patch. We only accept patches that increase the probability. A key advantage of using LLMs is that we can use additional context not available to existing neural bug detectors. As a result, our evaluation showed that our LLM-based validation

strategy with the help of extra file-level context can significantly reduce the false alarm rate of a neural bug detector while nearly maintaining its ability to detect and fix bugs. We believe that our validation approach can significantly improve the usability of existing neural bug detectors and thereby help developers to find bugs faster.

Recommendation. Based on our experimental results, we believe that in practice a hybrid strategy of testing and LLM-based validation might be the most effective for validating the output of neural bug detectors. Therefore, we recommend that developers use fast high-recall neural bug detectors to quickly identify potential bugs and their patches in the code base. Because testing is often costly for a large quantity of patches, we recommend using an LLM-based validator first and then execute tests on the remaining alarms and patches. We expect that alarms and patches that pass the validation and all tests likely indicate a real bug. To further increase the precision, equivalence checking [CPSA19] could be used in the future to filter out all semantics-preserving code changes.

7 Conclusion

The goal of this thesis was to systematically analyze and address potential limitations of neural bug detectors in (1) the training process and (2) the task design. To improve the training process, we developed a novel mutation strategy to generate more realistic training examples and we mined realistic bugs from public repositories. We systematically evaluated how more realistic mutants and real bug fixes impact the performance of neural bug detectors. To evaluate the impact of the task design, we proposed a validation strategy which can make use of context typically not available to a neural bug detector. In the following, we conclude this thesis by summarizing central contributions in Section 7.1. We further discuss them and provide an outlook for future work in Section 7.2.

7.1 Summary

This thesis builds upon two central hypotheses that have guided our research. We hypothesized that neural bug detectors that are historically only trained on mutated code are limited by their training process. To confirm this hypothesis and address this limitation, we developed two strategies.

Our first strategy and contribution of this thesis is to improve the mutation process with the help of a mutation operator that produces more realistic bugs. For this, we developed a novel contextual mutation operator that employs a masked language model to select more realistic mutations. We provided evidence that the mutants generated by our contextual mutation operator are more realistic than traditional mutations. Our experimental results showed that neural bug detectors trained on more realistic mutants are more effective in detecting and repairing real bugs.

Our second strategy that was motivated by this finding is to train neural bug detectors directly on real bug fixes. Real bug fixes represent previous bugs and their fixes provided by a developer. Real bug fixes are often difficult to obtain at larger

scales by mining open source repositories. Therefore, our second contribution is a mining process that we scaled to over 500K Git projects and the resulting collections of real bug fixes, `CTSSB-0.9M` and `CSSB-2.3M`. Our third contribution is a systematic evaluation of the impact of mutants and real bug fixes on the performance of neural bug detectors at different dataset scales. We found that neural bug detectors can still benefit from training on mutants even if real bug fixes are available. Furthermore, we showed that existing neural bug detectors are limited by their training process. By training neural bug detectors on significantly larger datasets of both mutants and real bug fixes, the performance of neural bug detectors on real bugs can be significantly improved. This confirms our first hypothesis.

Our second hypothesis is that neural bug detectors are limited by their task design. Existing neural bug detectors are often restricted to function-level neural bug detection. We hypothesized that this restriction is one of the key reason for the false alarms generated by a neural bug detector. Therefore, our fourth contribution is a validator that can validate the output of neural bug detectors based on extra file-level context not available to the neural bug detector. For this, we formulated the problem of validating the output of a neural bug detector as a patch validation problem and used large language models to validate the generated patches. The large language models can effectively utilize different contexts in this process. Therefore, when restricting the LLM-based validator to a function-level context, we found that this restriction mainly impacted the ability of the validator to identify false alarms. We see this finding as the first evidence to confirm our second hypothesis.

Based on these results, we conclude for our main research question:

> Existing neural bug detectors were limited by their training process and are limited by their task design. By training on more realistic examples, neural bug detectors can become significantly more effective in detecting and repairing real bugs. By using an external validator with extra context, the number of false alarms can be reduced significantly.

7.2 Discussion and Outlook

Finding and fixing software bugs is ongoing research problem in software engineering. Applying machine learning to learn from previous human mistakes and using validation to validate the output of bug detection tools are both likely interesting directions for future work. Our thesis reveals several ideas and lessons learned along the line which we discuss and highlight in the following.

Neural Bug Detectors as Hypothesis Generators. Our findings in Chapter 6 have motivated us to think a bit differently about neural bug detection. Instead of viewing existing neural bug detectors as an end-to-end solution, it can be beneficial to view them as a kind of *hypothesis generator*. Therefore, instead of directly relying on

the output of the neural bug detector, we could view the output as a *hypothesis* that the program contains a bug (or not) which could be fixed by the given patch. A central outcome of our results in Chapter 6 is that proving or disproving this hypothesis can be significantly easier than the original problem of bug detection. For example, while it is hard to show that a given program is bug-free via testing, it is significantly easier to show that a given patch is not bug-fixing (if it for example fails some tests). We believe that this slightly different view on the problem could open up new possibilities to address the problem of neural bug detection.

Learning from Bug Detection Models. By training on real bug fixes in Chapter 5, the neural bug detectors have become increasingly effective in detecting real bugs. However, the learned bug detection models lack any form of interpretability. Therefore, it is often unclear what the neural bug detectors have learned to become more effective at the task of bug detection. We believe that a better understanding of the learned behavior of neural bug detectors could potentially lead to new insights in how developers make mistakes and how they can be detected and fixed.

Cooperation with Formal Methods. Neural bug detectors learn to detect bugs by inferring an *implicit* correctness specification of code. For example, they can detect bugs in a `square(x)` function if the function does not compute the square of its input. However, because neural bug detectors are often based on probabilistic models, they cannot provide any formal guarantees over their output. Formal guarantees – especially in a security-critical context – can be highly important for trusting the output of the bug detection tool. Therefore, we could envision a *cooperation* between neural bug detector and formal software verifiers [JM09]. In contrast to neural bug detectors, formal software verifiers provide strong formal guarantees over their output. Software verifiers formally prove the correctness of software with respect to an *explicit* correctness specification. However, their application is limited by the need of an explicit formal specification. Therefore, we believe that a technology similar to neural bug detectors can be used to extract explicit formal specification from implicit correctness specifications provided in the program. As a result, we could use formal verification methods to detect bugs in programs without formal specifications.

We already performed a preliminary investigation on this topic in [JRW24]. Here, we investigated whether LLMs can infer formal properties (loop invariants) of code. In the future, it could be interesting to use LLMs for inferring formal specifications (program invariants) from code.

Implications for Software Engineering. By training on more advanced mutants and real bug fixes in Chapter 5, neural bug detectors have become significantly more effective in detecting (and repairing) real bugs. With the help of external validators in Chapter 6, the number of false alarms can be significantly reduced. For these reasons, we believe that neural bug detectors (among many other tools) can already be helpful for software developers to detect bugs earlier in the development process. In fact,

during our evaluation in Section 6.5.4, we discovered 20 new bugs in 16 popular Python projects with the help of a neural bug detector. However, to fully utilize the capabilities of neural bug detectors in the development process, there are several challenges that we should overcome in the future. For example, Sadowski et al. [SAE+18] reports that software developers often expect that bug detectors are deeply integrated in their workflow. Therefore, existing static bug detectors are often integrated into existing compiler toolchains. Because of the higher demand of computing resources and the demand for specialized hardware, e.g. GPUs, to run more efficiently, we believe that a similar integration of neural bug detectors might be more challenging. However, we can envision to run neural bug detectors as a pre-commit hook. Here, the neural bug detector is executed on the changed code before it goes into production code. The execution can be done on external server with specialized hardware.

Overall, we believe that neural bug detection represents an exciting research direction in software engineering which could make *automated* debugging methods more accessible for a wider audience of software developers.

A Appendix

A.1 Simple Stupid Bug Patterns

In the following, we provide a list of all "simple stupid bug" (SStuB) patterns as proposed by Karampatsis and Sutton [KS20] and later extended for Python by Kamienski et al. [KPBH21]:

- *Change Identifier Bugs*: The developer uses the wrong identifier to access a memory location or to call a function. The bug patterns includes the wrong usage of *variable names*, *parameter names* or *function names*. It is easy for the developer to utilize the wrong identifier with the same type. Similar identifier names (e.g. `patch` and `patches`) that appear in the same code might further contribute to the occurrence of this bug type.
- *Change Numeric Literal Bug*: The developer uses the wrong numeric literal. For example, the developer uses accidentally the second index of an array (1) even though she meant the first (0).
- *Change Boolean Literal Bug*: The developer uses the wrong boolean literal. For example, the developer returns accidentally `False` after successfully validating the input to the program.
- *Change Modifier Bug*: The developer uses the wrong modifier for variable, function, or class. This bug pattern is specific to Java.
- *Wrong Function Name Bug (or API Misuse Bug)*: The developer uses a function name different from the intended function name with the same parameter list. While this pattern can also be considered as a part of change identifier bugs, it is often interesting to consider this bug pattern distinctly. The bug pattern is particularly challenging to detect and fix since it requires access to the API defined for the project.

- *Same Function More Args*: The developer uses too many arguments in the function call. This can for example occur when a function is overloaded with multiple definitions.
- *Same Function Less Args*: The developer forgets to define an argument of a function. This can also occur when a function is overloaded with multiple definitions.
- *Same Function Change Caller Bug*: The developer might confuse two different variables with the same type when calling a function. Therefore, functions might be executed on the wrong object.
- *Same Function Swap Args Bug*: The software developer accidentally uses the function arguments in the wrong order. This can for example happen if there exist multiple variables with the same type that used in the function call.
- *Change Binary Operator Bug*: The developer confuses a binary operator. For example, the developer confuses `<=` with `<` in a loop condition resulting in an *off-by-one* error [BSS+20]. Bugs that fall in this bug pattern can be easily generated with mutation operators.
- *Change Unary Operator Bug*: The developer confuses a binary operator. For example, the developer might forget to use a negation operator (`!` in Java or `not` in Python).
- *Change Operand Bug*: The developer uses the wrong operand in a binary expression.
- *More Specific If*: The developer forgets to add an extra condition (with `&&` in Java or `and` in Python) to an if-condition.
- *Less Specific If*: The developer forgets a case where the if-condition should hold and adds an alternative condition (with `||` in Java or `or` in Python) to an if-condition.
- *Missing Throws Exception*: The developer forgets to add a throws clause to a Java method. This pattern is specific to Java.
- *Delete Throws Exception*: The developer uses too many throws clauses for a Java method. This pattern is specific to Java.

Kamienski et al. [KPBH21] found that some patterns are very specific to Java and that there are frequent bug patterns in Python that are not covered by these SStuBs. Therefore, they extended the list of SStuBs with following definitions specific to Python:

- *Change Attribute Used*: The developer uses the wrong attribute from an object. For example, `car.type` is used although `car.name` was meant.

- *Add Function Around Expression*: The developer forgets to put a function around a given expression. Since Python uses functions for type casts, it is more common to forget a function around an expression, i.e. `int_size = 5 / 3` instead of `int_size = int(5 / 3)`.
- *Add Elements to Iterable*: The developer forgets to add elements to a hard-coded iterable.
- *Change Keyword Argument Used*: The developer uses the wrong keyword argument when calling a function. For example, instead of calling `Car(type=bmw)` the call `Car(name=bmw)` is used.
- *Add Method Call*: The developer forgets to add a method call to a given expression. For example, instead of using `car.color() == 'red'` the expression `car == 'red'` is used.
- *Change Constant Type*: The developer uses the wrong constant type. For example, the construction year is defined by `car.year = 1987` although a string might be preferred, i.e `car.year = '1987'`.
- *Add Attribute Access*: The developer forgets an attribute access. For example, `car = 1987` instead of `car.year = 1987` is used.

A.2 Alternative Contextual Mutation Operators

For our evaluation in Section 5.4, we explore and compare different variants of *contextual* mutation operators. These mutation operators all utilize the surrounding context to infer mutations that likely generate realistic bugs. The main difference is the way how they identify realistic mutations.

A.2.1 Contextual Mutation Operators

We consider different types of contextual mutation operators that are based on recent studies on realistic mutation generation.

Generate and Rank. While we used a sampling distribution to define our contextual mutation operator in Chapter 3, we found that not all mutation operators utilize a well-defined sampling distribution. Therefore, to represent all considered mutation operators in a common framework, we employ a *generate-and-rank* based strategy [ABJS16] for generating mutants. For this, we first generate a list of mutation candidates and then rank them according to some (contextual) ranking function. Selected are those mutants that are ranked highest.

Generating Mutation Candidates. We generate mutation candidates with the help of traditional mutation operators. Since we focus in Section 5.4 on single token bug detection, we also focus here on *single token mutation*. Therefore, to generate a mutation candidate for a given program represented by $T = t_1, \ldots, t_n$, we (randomly) replace a single token t_l with another token r. Here, the token r is either defined in the scope ($r \in \{t, \ldots, t_n\}$) or coming from an external vocabulary ($r \in V$). For our evaluation, we exhaustively generate the set of all possible single token mutations (considered in Section 5.4):

$$M_T \subseteq \{\ \langle l, r\rangle \mid t_l \in T, r \in T \cup V \text{ and } t_l \neq r\}$$

Hence, all mutations in M_T represent variable misuses, binary operator, unary operator or literal bugs. Note that the set of single token mutation is always finite and limited by the number of tokens and the size of the vocabulary V.

Contextual Ranking Strategies. We consider four different contextual ranking strategies: (1) naturalness, (2) unnaturalness, (3) adversial and (4) backtranslation based ranking strategies.

Naturalness [RW22b]. This ranking strategy is based on the operator we proposed in Chapter 3. Here, we employ a masked language model (MLM) to rank all mutation candidates based on the surrounding mutation context. For this, we mask out the location l to be mutated and score r according to the masked language model (i.e. we score r with respect to $P_{MLM}(r \mid t_1 \ldots [M] t_{m+1} \ldots t_n)$). Therefore, mutations are higher ranked if they are assigned a higher likelihood by the language model. These

mutations are considered more *natural* than others. The main difference of our naturalness mutation operator as compared to Chapter 3 is that we train a masked language model specific for this task from scratch.

Unnaturalness [ABJS16]. Allamanis et al. [ABJS16] proposed to select the *least* likely mutation (according to a language model) to generate realistic mutations. Therefore, we use the same construction as for our naturalness based ranking strategy, but invert the ranking. Mutations are ranked higher if they are considered less natural according to our MLM.

Adversial [AJFB21]. Alternatively, Allamanis et al. [AJFB21] proposed to generate mutations that are *hard to detect* for a given neural bug detector. Therefore, we rank mutations Y of a given example X according to the loss $\mathcal{L}_{loc+rep}$ (see Section 2.3.2). The loss $\mathcal{L}_{loc+rep}$ measures to likelihood of a given neural bug detector to detect and repair the mutation in Y_i. We employ RealiT [RW22a] as the neural bug detectors used to compute the loss function. Mutations that are harder to detect by the neural bug detector are ranked higher.

Backtranslation [PP21]. Since we now have access to a large collection of real bug fixes for training (see Section 5.4), we can also utilize them for training a mutation operator. For this, we invert the set of real bug fixes and fine-tune our masked language model (MLM) on the inverted real bug fixes. We rank mutations according to the likelihood of the mutation assigned by our backtranslation model.

Training. A key part of our ranking strategies is an effective masked language model (MLM). As existing MLMs [DCLT19] are often large scale models trained for (sub-) token replacements, they are difficult to apply for single token mutation at a large scale (which is our goal in Section 5.4). Therefore, we train a significantly smaller MLM (28M parameters) specialized for our single token mutations. For this, the MLM employs the same Transformer based architecture as our neural bug detectors (see Section 2.3). During training, we mainly train the repair head to model the probability $P_{MLM}(t_m \mid t_1 \dots [M] t_{m+1} \dots t_n)$, i.e. the probability that [M] can be replaced by t_m. We use a large scale Github corpus containing around 90M function implementations[1] for training and we mask out tokens that correspond to one of our bug types (e.g. variables, literals or operators). During this process, we mutate 15% of the data (similar to Devlin et al. [DCLT19]) with a regular mutation operator to also train the localization head. This allows us to fine-tune the MLM more effectively on inverted real bug fixes (for creating our backtranslation based ranking strategy).

[1] We downloaded all Python repositories from Google BigQuery (`https://console.cloud.google.com/marketplace/details/github/github-repos`). We then parsed all Python files and extracted all top-level functions. We deduplicated the dataset with respect to our test and validation sets.

A.2.2 Evaluation

For our evaluation, we implemented the different mutation strategies, i.e. the different combinations of mutant generation and ranking strategy, for generating variable misuse, binary operator, unary operator and literal bugs in Python. We performed several experiments to answer the following research question:

Which mutation strategy produces the most realistic mutants?

Based on our evaluation, we then select the mutation strategy that produces the most realistic mutants for our evaluation in Section 5.4.

Evaluation Task

To compare the different mutation strategies, we consider the task of *reproducing real bugs*.

Real Bug Reproduction. To assess the ability of a mutation strategy to reproduce real bugs, we employ the validation set of real bug fixes from our experiments in Section 5.4. The validation set contains around 2K real bug fixes derived from open source projects. For our experiments, we invert the real bug fixes which leaves us with a dataset $D = \{(X_i, Y_i)\}_{i=1}^{n}$ mapping a fixed variant X_i to a buggy program Y_i. During our experiments, we are interested whether a given mutation operator $\mathcal{M}$ is able to reproduce the buggy program Y_i given the repaired program X_i.

Metric. To measure the effectiveness of the mutation strategies, we report the *average recall at k* (AR@k). Given a set $D = \{(X_i, Y_i)\}_{i=1}^{n}$ of fixed variants X_i and buggy programs Y_i, we measure AR@k as follows:

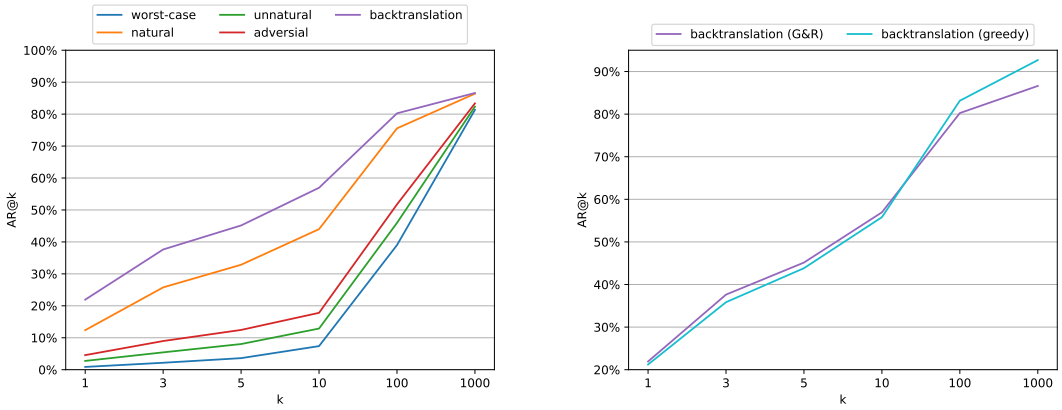
$$\text{AR@}k = \frac{1}{n}\sum_{i=1}^{n}[\![\text{rank}_i \leq k]\!],$$

where rank_i is the rank of Y_i (according to some ranking function) among all mutations of X_i generated by $\mathcal{M}$. To compute AR@k, we exhaustively generate mutation candidates of X_i and then rank them according to the different mutation strategies. We set $\text{rank}_i = \infty$ if Y_i cannot be generated via mutation. Note that AR@k measure the likelihood that the real buggy version Y_i of X_i is contained within the top-k mutants.

A.2.3 Which mutation strategy is more effective?

During our evaluation, we are interested which mutation strategy produces the most realistic mutations. For this, we assume that a mutation strategy that is most likely to reproduce real bugs also produces the most realistic mutations.

Experimental Setup. During our experiments, we consider the top-k mutants as generated by each mutation strategy. Based on these selection of mutants, we compute



(a) Mutation Strategies

(b) Generation Strategies

Figure A.1: Effect of mutation and generation strategy on the ability to reproduce real bugs on our validation set.

the AR@k score. Similar to our experiments in Section 5.4, we compute the AR@k score at $k \in \{1, 3, 5, 10, 100, 1000\}$. We prefer a mutation operator over another if its AR@k score is consistently higher across all k's.

Results. Our results are shown in Fig. A.1a. The x-axis reports the number k of mutants that are selected and the y-axis shows the respective AR@k score. As a baseline, we also report the worst-case AR@k performance which considers a mutation operator that always ranks Y_i last among all mutants of X_i.

Backtranslation works best for reproducing real bugs. Unsurprisingly, we find that the backtranslation based mutation strategy which is trained to reproduce real bugs is also most effective in reproducing real bugs. More surprising is the relatively small gap between the backtranslation based and the naturalness based mutation strategy. Therefore, especially in this comparison, the naturalness based mutation strategy seems to be quite effective in mutation generation when real bug fixes for training are not available. The adversial and the unnaturalness based mutation strategies perform significantly worse.

Around 15% of the real bugs cannot be reproduced. We find that there is an upper limit on the effectiveness of the mutation strategies. As a result, around 15% of the real bugs cannot be generated by any of the mutation strategies. After closer inspection, we find that the mutants are not generated by our traditional mutation operators: As the traditional mutation operators are designed to produce syntactically correct mutants, the mutation operator often exclude transformation that might lead to syntactical errors. An example is provided in Table A.1. Here, the developer confuses the assignment operator `=` with the equal comparison operator `==`. This type of mutation might trigger in some contexts a syntactical error. Therefore, traditional mutation operators often exclude this type of mutation.

Table A.1: Example of a software bug that cannot be reproduced via traditional mutation. Code is reformatted to fit the figure.

Example	Description
1 `# Assignment Operator Bug` 2 `writer = None` 3 `if file_format.lower() == 'ply':` 4 `  writer == vtk.vtkPLYWriter()` 5 `else:` 6 `  writer = vtk.vtkPolyDataWriter()`	The developer confuses `=` with a comparison operator `==`. The Python interpreter does not warn the developer as the use of `==` is syntactically correct. **Fix:** replace `==` in Line 4 by `=`.

To overcome this limitation, we experiment with greedily generated mutants directly based on the backtranslation model. As the backtranslation model assigns a probability to every possible single token mutation, it can in theory generate mutations not covered by traditional mutation operators. We compare greedy generation with the generate-and-rank based solution. The results of the comparison are shown in Fig. A.1b. Although greedy generation does not significantly improve the reproduction performance over generate-and-rank, we find that the backtranslation based mutation strategy with greedy generation is able to reproduce real bugs that traditional mutation operator typically cannot.

Based on our observations, we conclude:

> The backtranslation based mutation strategy is most effective in reproduce real bugs within k mutations (based on AR@k). As we expect (based on our results in Chapter 3) that neural bug detectors trained on more realistic mutants are more effective on detecting and repairing real bugs, we will utilize the backtranslation based mutation operator with a greedy generation strategy in our experiments in Section 5.4.

A.3 Scanning Open Source Projects for Real Bugs

In this section, we provide more details on the projects that we scanned in our evaluation in Section 6.5.4 and on the bugs and quality issues that we found in the process.

A.3.1 Scanned Projects

In the following, we provide an overview of all open source projects that we scanned for our evaluation in Section 6.5.4. We report in Table A.2 the project name, the creation time (in format of days/month/year), the number of stars (at the time of scanning the repository), the number of functions in the project and the number of alarms raised by our neural bug detector (before validation).

Table A.2: List of all Open Source Projects scanned by our neural bug detector.

Project	Creation Time	# Stars	# Functions	# Alarms
Significant-Gravitas/AutoGPT	16/03/2023	169K	2K	468
abi/screenshot-to-code	14/11/2023	63K	60	22
xtekky/gpt4free	29/03/2023	62K	717	220
OpenInterpreter/open-interpreter	14/07/2023	57K	515	142
meta-llama/llama	14/02/2023	57K	27	7
zylon-ai/private-gpt	02/05/2023	54K	252	55
gpt-engineer-org/gpt-engineer	29/04/2023	53K	370	71
xai-org/grok-1	17/03/2024	50K	86	33
geekan/MetaGPT	30/06/2023	46K	3K	880
oobabooga/text-generation-webui	21/12/2022	41K	967	319
THUDM/ChatGLM-6B	13/03/2023	41K	110	36
Stability-AI/stablediffusion	23/11/2022	39K	698	276
lm-sys/FastChat	19/03/2023	37K	1K	441
run-llama/llama_index	02/11/2022	37K	14K	2K
QuivrHQ/quivr	12/05/2023	37K	287	69
RVC-Boss/GPT-SoVITS	14/01/2024	36K	1K	457
XingangPan/DragGAN	18/05/2023	36K	1K	487
microsoft/autogen	18/08/2023	35K	2K	446
chenfei-wu/TaskMatrix	02/03/2023	35K	109	40
2noise/ChatTTS	27/05/2024	33K	423	156
Pythagora-io/gpt-pilot	16/08/2023	32K	761	160
vllm-project/vllm	09/02/2023	31K	9K	3K
lllyasviel/ControlNet	01/02/2023	31K	2K	754
tatsu-lab/stanford_alpaca	10/03/2023	30K	22	6
s0md3v/roop	28/05/2023	29K	86	34
meta-llama/llama3	15/03/2024	27K	38	15
svc-develop-team/so-vits-svc	10/03/2023	26K	1K	479
Vision-CAIR/MiniGPT-4	15/04/2023	25K	603	193
Stability-AI/generative-models	22/06/2023	25K	658	287
roboflow/supervision	28/11/2022	24K	785	188
infiniflow/ragflow	12/12/2023	24K	2K	461
microsoft/JARVIS	30/03/2023	24K	171	80
Aider-AI/aider	09/05/2023	23K	947	227
AIHawk-FOSS/Auto_Jobs_Applier_AI_Agent	04/08/2024	23K	203	49
crewAIInc/crewAI	27/10/2023	22K	1K	225
facebookresearch/audiocraft	08/06/2023	21K	1K	394
yoheinakajima/babyagi	03/04/2023	20K	243	37
facefusion/facefusion	17/08/2023	20K	979	330
microsoft/graphrag	27/03/2024	20K	1K	161
mlc-ai/mlc-llm	29/04/2023	19K	2K	789
opendatalab/MinerU	29/02/2024	19K	800	232
unslothai/unsloth	29/11/2023	19K	285	114
ymcui/Chinese-LLaMA-Alpaca	15/03/2023	18K	84	34
NanmiCoder/MediaCrawler	09/06/2023	18K	639	80
VikParuchuri/marker	30/10/2023	18K	283	90
black-forest-labs/flux	01/08/2024	18K	122	53
Mikubill/sd-webui-controlnet	12/02/2023	17K	6K	2K
apple/ml-stable-diffusion	16/11/2022	17K	177	78
huggingface/peft	25/11/2022	17K	2K	746
openai/swarm	22/02/2024	16K	157	34

A.3.2 Found Bugs and Quality Issues

In the following, we provide an overview of the found bugs and quality issues we found in open source projects. The code is abbreviated to fit the figures. We omit bugs and quality issues that we found multiple times across projects (i.e. `==` comparisons with `None`).

Example	Description

Confirmed bug in run-llama/llama_index
`client = None`
`if provided_client is not None:`
`-  client = client`
`+  client = provided_client`
	The implementation checks if a client is provided but does not use the provided client.
	Fix: Replace `client` with `provided_client`.

Example	Description

Potential bug in oobabooga/text-generation-webui
`metadata = body.get('metadata')`
`- if corpus is None:`
`+ if metadata is None:`
`  self._send_412_error("Missing parameter 'metadata'")`
`  return`
	Copy paste bug. The code should check whether `metadata` is defined but checks for the `corpus`.
	Fix: Replace `corpus` with `metadata`.

Example	Description

Potential bug in run-llama/llama_index
`odata_filter = None`
`# NOTE: users can provide odata_filters directly to the query`
`odata_filters = kwargs.get("odata_filters")`
`if odata_filters is not None:`
`- odata_filter = odata_filter`
`+ odata_filter = odata_filters`
`else:`
`  if query.filters is not None:`
`  odata_filter = self._create_odata_filter(query.filters)`
	If `odata_filters` is given, the current `odata_filter` should be overwritten. However, the developer uses `odata_filter` accidentally. Hence, defining `odata_filters` has no effect on the implementation.
	Fix: Replace `odata_filter` with `odata_filters`.

Example	Description

Potential bug in THUDM/ChatGLM-6B
`if isinstance(unwrap_model(self.model), PreTrainedModel):`
`  if state_dict is None:`
`  state_dict = self.model.state_dict()`
`  unwrap_model(self.model).save_pretrained(`
`- output_dir, state_dict=filtered_state_dict)`
`  unwrap_model(self.model).save_pretrained(`
`+ output_dir, state_dict=state_dict)`
`  else:`
	Copy paste bug. `filtered_state_dict` is defined in a different branch and hence undefined here. As the developer loads the `state_dict` shortly before saving, `state_dict` should likely be saved.
	Fix: Replace `filtered_state_dict` with `state_dict`.

Example	Description

Potential bug in run-llama/llama_index
`guidance_llm: Optional["GuidanceLLM"] = None,`
`verbose: bool = False,`
`):`
`- if not guidance_llm:`
`+ if guidance_llm:`
`  llm = guidance_llm`
`  else:`
`  llm = OpenAI("gpt-3.5-turbo")`
	The guidance LLM is never used because the condition checks if `guidance_llm` is not given.
	Fix: Replace `not guidance_llm` with `guidance_llm`.

Example	Description

Potential bug in huggingface/peft
`  if inference:`
`  config["inference_mode"] = True`
`  config_dict[key] = config`
`- return config`
`+ return config_dict`
	The function should generate a `config_dict`. However, the function returns a `config`. `config_dict` is defined but not used.
	Fix: Replace `config` with `config_dict`.

Example	Description

Potential bug in microsoft/JARVIS
`while True:`
`  try:`
`  result = chain.run(question=question,`
`- Too_list=Tool_dic)`
`+ Too_list=Tool_list)`
`  clean_answer = eval(result.split("(")[0].strip())`
	The function formats a set of tools that can be used by an LLM. However, the tool list is never provided to the LLM.
	Fix: Replace `Tool_dic` with `Tool_list`.

Example	Description

Potential bug in vllm-project/vllm
`  chunk_ids = prompt[idx:idx + chunk_size]`
`- data = SequenceData.from_seqs(prompt)`
`+ data = SequenceData.from_seqs(chunk_ids)`
`  data.update_num_computed_tokens(idx)`
`  seq_data = {i: data}`
`  seq_group_metadata_list.append(`
	The function prepares a chunk of prompts, but then uses the complete sequence of prompts.
	Fix: Replace `prompt` with `chunk_ids`.

Example	Description

Potential bug in microsoft/graphrag
`  attributes = covariates[0].attributes or {}`
`  if len(covariates) > 0 else {}`
`  attribute_cols = list(attributes.keys())`
`-  if len(covariates) > 0 else []`
`  attribute_cols = list(attributes.keys())`
`+  if len(attributes) > 0 else []`
	Copy paste bug. The code should likely check if `attributes` is non-empty to then compute the `attribute_cols`. However, `covariates` is used instead.
	Fix: Replace `covariates` with `attributes`.

Example	Description

Potential bug in lm-sys/FastChat
`nsfw_flag, csam_flag = image_moderation_filter(img)`
`self.assertFalse(nsfw_flag)`
`- self.assertFalse(nsfw_flag)`
`+ self.assertFalse(csam_flag)`
	Copy paste bug. The test checks the `nsfw_flag` twice. Likely the `csam_flag` should be checked instead.
	Fix: Replace `nsfw_flag` with `csam_flag`.

Example	Description

Potential bug in QuivrHQ/quivr
`list_files = [file.file_name or file.url for`
`  file in list_files_array]`
`files = list(filter(lambda n: n is not None,`
`  list_files))`
`files = files[:max_files]`
 	
`files_str = "\n".join(files)`
`-  if list_files_array else "None"`
`files_str = "\n".join(files)`
`+  if files else "None"`
`return files_str`
	Even if `list_files_array` is not empty, `files` might be empty. Therefore, the developer likely wants to check for the emptiness of `files`.
	Fix: Replace `list_files_array` with `files`.

Example	Description

Potential bug in xtekky/gpt4free
`system_message = "\n".join(...)`
`- if system_message:`
`+ if not system_message:`
`  system_message = "A chat between ..."`
	The `system_message` is always overridden if a `system_message` is provided. Likely, the `system_message` should only be overridden with a default message if no `system_message` is provided.
	Fix: Replace `system_message` with `not system_message`.

Example	Description

Potential bug in zylon-ai/private-gpt
`result = subprocess.run(...)`
`- assert result.returncode != 0, f"Script failed with error: {result.stderr}"`
`+ assert result.returncode == 0, f"Script failed with error: {result.stderr}"`
	If we check the error message, it becomes clear that the subprocess should succeed. However, the `assert` fails only if the subprocess succeeds (`returncode != 0`).
	Fix: Replace `!=` with `==`.

Example	Description

Potential bug in microsoft/graphrag
`- return self._parse_claim_tuples(results, prompt_args)`
`+ return self._parse_claim_tuples(claims, prompt_args)`
	The code preprocesses `results` to add additional claims. However, the function that processes the generated claims is only provided with the unprocessed `results`.
	Fix: Replace `results` with `claims`.

Example	Description

Potential bug in infiniflow/ragflow
`- while len(long_string) <= NAME_LIMIT:`
`+ while len(long_string) < NAME_LIMIT:`
`  long_string += random.choice(string.ascii_letters + string.digits)`
	The code should likely generate a name with `NAME_LIMIT` characters. However, the code guarantees that the name is always longer than the `NAME_LIMIT`.
	Fix: Replace `<=` with `<`.

Example	Description

Potential bug in lllyasviel/ControlNet
`output_height_i=out_h,`
`- output_width_i=out_h,`
`+ output_width_i=out_w,`
`spatial_scale_f=spatial_scale,`
	In most cases it holds that `out_h == out_w`. However, there exists cases where `out_h != out_w` which is not covered by the code.
	Fix: Replace `out_h` with `out_w`.

Example	Description

Potential bug in mlc-ai/mlc-llm
`engine_config=EngineConfig(`
`- max_num_sequence=max_history_size,`
`+ max_num_sequence=max_num_sequence,`
`max_total_sequence_length=max_total_sequence_length,`
`prefill_chunk_size=prefill_chunk_size,`
`max_history_size=max_history_size,`
	The user can specify `max_num_sequence`. However, the code currently ignores the value of `max_num_sequence`.
	Fix: Replace `max_history_size` with `max_num_sequence`.

Example	Description

Potential bug in svc-develop-team/so-vits-svc
`temp = torch.arange(src_len+1) * target_len`
`-  / src_len`
`temp = torch.arange(src_len+1) * target_len`
`+  // src_len`
`current_pos = 0`
	In earlier version of PyTorch, the `/` operator would perform a floor division which is correct here. However, in newer version `/` might generate a floating point number which is not expected here.
	Fix: Replace `/` with `//`.

Example	Description

Potential bug in svc-develop-team/so-vits-svc
`  postfix='',`
`  device='cpu'):`
`- if postfix == '':`
`+ if postfix != '':`
`  postfix = '_' + postfix`
`path = os.path.join(expdir, name+postfix)`
	The separator `_` should likely be used to separate the name from the `postfix`. However, the separator is only used when the postfix is empty.
	Fix: Replace `==` with `!=`.

Example	Description

Potential bug in Stability-AI/generative-models
`1 state = dict()`
`2 - if not "model" in state:`
`3 + if not "model" in version_dict:`
`4 config = version_dict["config"]`
`5 ckpt = version_dict["ckpt"]`
	The check always succeeds since `state` is defined to be empty.
	Fix: Replace `state` with `version_dict`.

Example	Description

Potential quality issue in OpenInterpreter/open-interpreter
`1 async def output(self):`
`2 - if self.output_queue == None:`
`3 + if self.output_queue is None:`
`4 self.output_queue = janus.Queue()`
`5 return await self.output_queue.async_q.get()`
	The comparison with `None` using the `==` operator can have unwanted consequences. The Python standard recommends to use `is` always when comparing with `None`.
	Fix: Replace `==` with `is`.

Example	Description

Potential quality issue in chenfei-wu/TaskMatrix
`1 merged_mask_image = Image.fromarray(`
`→ merged_mask)`
`2 - return merged_mask`
	The variable `merged_mask_image` is unused which might indicate a quality issue. The neural bug detector flags the return statement which is incorrect here. However, an alarm here might still be useful.

Example	Description

Potential quality issue in RVC-Boss/GPT-SoVITS
`1 reject_y = torch.stack(reject_y, dim = 0)`
`2 reject_y_lens = torch.tensor(reject_y_lens,`
`3 - device=y_lens.device)`
`4 reject_y_lens = torch.tensor(reject_y_lens,`
`5 + device=y_o.device)`
`6 return reject_y, reject_y_lens`
	In most cases, `y_lens.device == y_o.device`. However, to be more consistent with the previous code, it can make sense to use `y_o` here.
	Fix: Replace `y_lens` with `y_o`.

Example	Description

Potential quality issue in Stability-AI/generative-models
`1 - if "fps" not in ukeys:`
`2 + if "fps" not in value_dict:`
`3 value_dict["fps"] = 6`
`4 value_dict["is_image"] = 0`
	There exists an implicit dependency between `ukeys` and `value_dict`: If fps is defined in `ukeys`, then it is also defined in `value_dict`. Therefore, the change is semantics-preserving. However, applying the fix could still improve the coding style.
	Fix: Replace `ukeys` with `value_dict`.

Example	Description

Potential quality issue in OpenInterpreter/open-interpreter
`1 matches = [match.group() for match in ...]`
`2 - matches += [match.replace("\\", "")`
`3 for match in matches if match]`
`4 + matches = [match.replace("\\", "")`
`5 for match in matches if match]`
`6 existing_paths = [match for match in matches`
`→ if os.path.exists(match)`
`→ ]`
`7 return max(existing_paths, key=len) if`
`→ existing_paths else None`
	The code should likely preprocess the matches. However, it actually adds the preprocessed matches to the original patches. Due to the way matches are processed, it might not matter for the implementation.
	Fix: Replace `+=` with `=`.

Bibliography

[ABDS18] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37, 2018.

[ABJS16] Miltiadis Allamanis, Earl T. Barr, René Just, and Charles Sutton. Tailored Mutants Fit Bugs Better. *CoRR*, abs/1611.02516, 2016.

[ABK17] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[ABK18] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[AHM+08] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William W. Pugh. Using Static Analysis to Find Bugs. *IEEE Softw.*, 25(5):22–29, 2008.

[AHO24] Kamel Alrashedy, Vincent J. Hellendoorn, and Alessandro Orso. Learning Defect Prediction from Unrealistic Data. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*, pages 556–567. IEEE, 2024.

[AJFB21] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876, 2021.

[AL21] Abdulaziz Alaboudi and Thomas D. LaToza. An Exploratory Study of Debugging Episodes. *CoRR*, abs/2105.02162, 2021.

[All70] Frances E. Allen. Control flow analysis. In Robert S. Northcote, editor, *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*, pages 1–19. ACM, 1970.

[All19] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In Hidehiko Masuhara and Tomas Petricek, editors, *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*, pages 143–153. ACM, 2019.

[AP10] Nathaniel Ayewah and William W. Pugh. The Google FindBugs fixit. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 241–252. ACM, 2010.

[Art] Charles Arthur. Apple's SSL iPhone vulnerability: how did it happen, and what next? `https://www.theguardian.com/technology/2014/feb/25/apples-ssl-iphone-vulnerability-how-did-it-happen-and-what-next`. Accessed: 2024-11-20.

[ASPK12] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*, pages 14–23. IEEE Computer Society, 2012.

[BB84] D Bartram and R Bayliss. Automated testing: Past, present and future. *Journal of Occupational Psychology*, 57(3):221–237, 1984.

[BBC+10] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[BFHORQ10] Antonio Bella, Cèsar Ferri, José Hernández-Orallo, and María José Ramírez-Quintana. Calibration of machine learning models. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, pages 128–146. IGI Global, 2010.

[BJT+22] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient Train-

ing of Language Models to Fill in the Middle. *CoRR*, abs/2207.14255, 2022.

[BKP24] Islem Bouzenia, Bajaj Piyush Krishan, and Michael Pradel. DyPyBench: A Benchmark of Executable Python Software. *Proc. ACM Softw. Eng.*, 1(FSE):338–358, 2024.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[Bri50] Glenn W Brier. Verification of forecasts expressed in terms of probability. *Monthly weather review*, 78(1):1–3, 1950.

[BSPC19a] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA):159:1–159:27, 2019.

[BSPC19b] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[BSS+20] Jón Arnar Briem, Jordi Smit, Hendrig Sellik, Pavel Rapoport, Georgios Gousios, and Maurício Aniche. OffSide: Learning to Identify Mistakes in Boundary Conditions. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 203–208, New York, NY, USA, 2020. Association for Computing Machinery.

[BWH22a] Nghi Bui, Yue Wang, and Steven C. H. Hoi. Detect-localize-repair: A unified framework for learning to debug with codet5. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 812–823. Association for Computational Linguistics, 2022.

[BWH22b] Nghi DQ Bui, Yue Wang, and Steven Hoi. Detect-localize-repair: A unified framework for learning to debug with codet5. *arXiv preprint arXiv:2211.14875*, 2022.

[C+90] IEEE Standards Committee et al. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610:12, 1990.

[CDAR20] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 2020.

[CDD+15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving Fast with Software Verification. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015.

[Cho14] Noam Chomsky. *Aspects of the Theory of Syntax*. Number 11. MIT press, 2014.

[CKT+21] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Software Eng.*, 47(9):1943–1959, 2021.

[CPSA19] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1027–1040. ACM, 2019.

[CQT+24] Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. LongLoRA: Efficient Fine-tuning of Long-Context Large Language Models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[Cro] External Technical Root Cause Analysis — Channel File 291. `https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf`. Accessed: 2024-11-20.

[CTJ+21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374, 2021.

[CZN+23] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[DHK+16] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-Scale Automated Vulnerability Addition. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 110–121. IEEE Computer Society, 2016.

[DP22] Renzo Degiovanni and Mike Papadakis. μbert: Mutation testing using pre-trained language models. In *15th IEEE International Conference on Software Testing, Verification and Validation Workshops ICST Workshops 2022, Valencia, Spain, April 4-13, 2022*, pages 160–169. IEEE, 2022.

[DZT+23] Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. Large Language Models of Code Fail at Completing Code with Potential Bugs. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[FAL+23] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[FMB+14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324. ACM, 2014.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[GDPT24] Aayush Garg, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. On the coupling between vulnerabilities and llm-generated mutants: A study on vul4j dataset. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2024, Toronto, ON, Canada, May 27-31, 2024*, pages 305–316. IEEE, 2024.

[GLLL16] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O. K. Li. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

[GPKS17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common C language errors by deep learning. In *Thirty-First AAAI conference on artificial intelligence*, 2017.

[GR07] Tilmann Gneiting and Adrian E Raftery. Strictly proper scoring rules, prediction, and estimation. *Journal of the American statistical Association*, 102(477):359–378, 2007.

[GVS+19] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. BugsJS: a Benchmark of JavaScript Bugs. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 90–101. IEEE, 2019.

[HBS+12] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 837–847. IEEE Computer Society, 2012.

[HBV22] Jingxuan He, Luca Beurer-Kellner, and Martin T. Vechev. On Distribution Shift in Learning-based Bug Detectors. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 8559–8580. PMLR, 2022.

[HD17] Vincent J. Hellendoorn and Premkumar T. Devanbu. Are deep neural networks the best choice for modeling source code? In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 763–773. ACM, 2017.

[HNP09] Alon Y. Halevy, Peter Norvig, and Fernando Pereira. The Unreasonable Effectiveness of Data. *IEEE Intell. Syst.*, 24(2):8–12, 2009.

[HP04] David Hovemeyer and William W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[HP18] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 317–328. ACM, 2018.

[HSS+19] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2019.

[HV23] Jingxuan He and Martin T. Vechev. Large language models for code: Security hardening and adversarial testing. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 1865–1879. ACM, 2023.

[HVKV24] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin T. Vechev. Instruction Tuning for Secure Code Generation. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[HWG+19] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR*, abs/1909.09436, 2019.

[HYC+24] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder Technical Report. *CoRR*, abs/2409.12186, 2024.

[HZ13] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors, *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 121–130. IEEE Computer Society, 2013.

[Inf] Website of Infer. `https://fbinfer.com`. Accessed: 2024-11-20.

[Jac12] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912.

[JADM23] Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. Large Language Models and Simple, Stupid Bugs. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*, pages 563–575. IEEE, 2023.

[JJE14] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs.

In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM, 2014.

[JJI+14] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 654–665. ACM, 2014.

[JKA17] René Just, Bob Kurtz, and Paul Ammann. Inferring mutant utility from program context. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 284–294. ACM, 2017.

[JLLT23] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1430–1442. IEEE, 2023.

[JLT21a] Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: code-aware neural machine translation for automatic program repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1161–1173. IEEE, 2021.

[JLT21b] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[JLZ+22] Yanjie Jiang, Hui Liu, Yuxia Zhang, Weixing Ji, Hao Zhong, and Lu Zhang. Do bugs lead to unnaturalness of source code? In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1085–1096. ACM, 2022.

[JM00] Daniel Jurafsky and James H. Martin. *Speech and language processing - an introduction to natural language processing, computational linguistics, and speech recognition.* Prentice Hall series in artificial intelligence. Prentice Hall, 2000.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, 2009.

[JRW24] Christian Janßen, Cedric Richter, and Heike Wehrheim. Can ChatGPT support software verification? In Dirk Beyer and Ana Cavalcanti, editors, *Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings*, volume 14573 of *Lecture Notes in Computer Science*, pages 266–279. Springer, 2024.

[JSMB13] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 672–681. IEEE Computer Society, 2013.

[Jus14] René Just. The major mutation framework: efficient and scalable mutation analysis for java. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 433–436. ACM, 2014.

[KAL22] Hong Jin Kang, Khai Loong Aw, and David Lo. Detecting false alarms from automatic static analysis tools: How far are we? In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 698–709. ACM, 2022.

[Kat20] Jeremy Katz. Libraries.io Open Source Repository and Dependency Metadata, 2020.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[KB19] Thommen George Karimpanal and Roland Bouffanais. Self-organizing maps for storage and transfer of knowledge in reinforcement learning. *Adapt. Behav.*, 27(2), 2019.

[KBR+20] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code != big vocabulary: open-vocabulary

models for source code. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1073–1085. ACM, 2020.

[KBZ+20] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning. In *European conference on computer vision*, pages 491–507. Springer, 2020.

[KDPT23] Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Efficient Mutation Testing via Pre-Trained Language Models. *CoRR*, abs/2301.03543, 2023.

[KMBS20] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Pre-trained Contextual Embedding of Source Code. *CoRR*, abs/2001.00059, 2020.

[KMJ+22a] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement, and Neel Sundaresan. Learning to Reduce False Positives in Analytic Bug Detectors. *arXiv preprint arXiv:2203.09907*, 2022.

[KMJ+22b] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin B. Clement, and Neel Sundaresan. Learning to Reduce False Positives in Analytic Bug Detectors. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1307–1316. ACM, 2022.

[KPBH21] Arthur V. Kamienski, Luisa Palechor, Cor-Paul Bezemer, and Abram Hindle. Pysstubs: Characterizing single-statement bugs in popular open-source python projects. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 520–524. IEEE, 2021.

[KS20] Rafael-Michael Karampatsis and Charles Sutton. How Often Do Single-Statement Bugs Occur?: The ManySStuBs4J Dataset. In Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 573–577. ACM, 2020.

[KW17] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[KXLL16] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 165–176. ACM, 2016.

[LAR17] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 727–739, 2017.

[LGR+21] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. In Joaquin Vanschoren and Sai-Kit Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021.

[LLH+24] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the Middle: How Language Models Use Long Contexts. *Trans. Assoc. Comput. Linguistics*, 12:157–173, 2024.

[LPP+20a] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In Sarfraz Khurshid and Corina S. Pasareanu, editors, *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 101–114. ACM, 2020.

[LPP+20b] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[LWN20] Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614, 2020.

[LWWM22] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. Context-Aware Code Change Embedding for Better Patch Correctness Assessment. *ACM Trans. Softw. Eng. Methodol.*, 31(3):51:1–51:29, 2022.

[MDF+01] José Carlos Maldonado, Márcio Eduardo Delamaro, Sandra CPF Fabbri, Adenilso da Silva Simão, Tatiana Sugeta, Auri Marcelo Rizzo Vincenzi, and Paulo Cesar Masiero. Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation testing for the new century*, pages 113–116. Springer, 2001.

[MFG24] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. *CoRR*, abs/2404.07143, 2024.

[MLZ+23] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. OctoPack: Instruction Tuning Code Large Language Models. *CoRR*, abs/2308.07124, 2023.

[MMY+21] Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. Megadiff: A dataset of 600k java source code changes categorized by diff size. *CoRR*, abs/2108.04631, 2021.

[MSC+23] Seungjun Moon, Yongho Song, Hyungjoo Chae, Dongjin Kang, Taeyoon Kwon, Kai Tzu-iunn Ong, Seung-won Hwang, and Jinyoung Yeo. Coffee: Boost Your Code LLMs by Fixing Bugs with Feedback. *CoRR*, abs/2311.07215, 2023.

[NOP+23] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. VULGEN: realistic vulnerability generation via pattern mining and deep learning. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2527–2539. IEEE, 2023.

[NPH+23] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[Off92] A. Jefferson Offutt. Investigations of the Software Testing Coupling Effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.

[PC22] Michael Pradel and Satish Chandra. Neural software analysis. *Commun. ACM*, 65(1):86–96, 2022.

[PKJ+21] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. In Joaquin Vanschoren and Sai-Kit Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021.

[PKZ+19] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.*, 112:275–378, 2019.

[PP21] Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 906–918. ACM, 2021.

[PR23] Julian Aron Prenner and Romain Robbes. RunBugRun - An Executable Dataset for Automated Program Repair. *CoRR*, abs/2304.01102, 2023.

[PR24] Julian Aron Prenner and Romain Robbes. Out of Context: How important is Local Context in Neural Program Repair? In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 83:1–83:13. ACM, 2024.

[PS18] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[QLAR15] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[R+62] Frank Rosenblatt et al. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*, volume 55. Spartan books Washington, DC, 1962.

[RBV16] Veselin Raychev, Pavol Bielik, and Martin T. Vechev. Probabilistic model for code with decision trees. In Eelco Visser and Yannis Smarag-

dakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 731–747. ACM, 2016.

[RGG+23] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code. *CoRR*, abs/2308.12950, 2023.

[RHG+16] Baishakhi Ray, Vincent J. Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar T. Devanbu. On the "naturalness" of buggy code. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 428–439. ACM, 2016.

[RHJ+22] Cedric Richter, Jan Haltermann, Marie-Christine Jakobs, Felix Pauck, Stefan Schott, and Heike Wehrheim. Are neural bug detectors comparable to software developers on variable misuse bugs? In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 9:1–9:12. ACM, 2022.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[RPDH18] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: challenging bug-finding tools with deep faults. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 224–234. ACM, 2018.

[RW22a] Cedric Richter and Heike Wehrheim. Can we learn from developer mistakes? learning to localize and repair real bugs from real bug fixes. *CoRR*, abs/2207.00301, 2022.

[RW22b] Cedric Richter and Heike Wehrheim. Learning Realistic Mutations: Bug Creation for Neural Bug Detectors. In *2022 IEEE Conference on Soft-*

ware Testing, Verification and Validation (ICST), pages 162–173. IEEE, 2022.

[RW22c] Cedric Richter and Heike Wehrheim. TSSB-3M: Mining single statement bugs at massive scale. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 418–422. ACM, 2022.

[RW23] Cedric Richter and Heike Wehrheim. How to Train Your Neural Bug Detector: Artificial vs Real Bugs. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 1036–1048. IEEE, 2023.

[SAE+18] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at Google. *Commun. ACM*, 61(4):58–66, 2018.

[SBLB15] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 532–543. ACM, 2015.

[SFM23] André Silva, Sen Fang, and Martin Monperrus. Repairllama: Efficient representations and fine-tuned adapters for program repair. *CoRR*, abs/2312.15698, 2023.

[SFYM23] André Silva, João F. Ferreira, He Ye, and Martin Monperrus. MUFIN: Improving Neural Repair Models with Back-Translation. *CoRR*, abs/2304.02301, 2023.

[SFZ11] Haihao Shen, Jianhong Fang, and Jianjun Zhao. EFindBugs: Effective Error Ranking for FindBugs. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 299–308. IEEE Computer Society, 2011.

[SHB16] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

[SLL+18] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. Bugs.jar: a large-scale, diverse dataset of real-world Java bugs.

In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 10–13. ACM, 2018.

[SLM17] Abigail See, Peter J. Liu, and Christopher D. Manning. Get To The Point: Summarization with Pointer-Generator Networks. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1073–1083. Association for Computational Linguistics, 2017.

[Spe61] Charles Spearman. The proof and measurement of association between two things. 1961.

[SUV18] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-Attention with Relative Position Representations. In Marilyn A. Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pages 464–468. Association for Computational Linguistics, 2018.

[TLK+20] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 981–992. IEEE, 2020.

[TMS+23] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela

Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR*, abs/2307.09288, 2023.

[TPW+19] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 25–36. IEEE, 2019.

[TTH+22] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F. Bissyandé. Is this change the answer to that problem?: Correlating descriptions of bug and code changes for evaluating patch correctness. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 59:1–59:13. ACM, 2022.

[TWB+19a] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–312. IEEE, 2019.

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

[TWB+19c] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, 2019.

[Twi] Twitter outage due to Formatting bug. `https://news.ycombinator.com/item?id=8810157`. Accessed: 2024-11-20.

[vD14] Arie van Deursen. Learning from Apple's# gotofail security bug. *Arie van Deursen: Software Engineering in Theory and Practice*, 2014.

[VFJ15] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer Networks. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2692–2700, 2015.

[VKM+19] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720*, 2019.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[WDS+19] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *CoRR*, abs/1910.03771, 2019.

[WDS+20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[Wer82] P J Werbos. Applications of advances in nonlinear sensitivity analysis, Jan 1982.

[WG24] David Gray Widder and Claire Le Goues. What is a" bug"? On subjectivity, epistemic power, and implications for software research. *arXiv preprint arXiv:2402.08165*, 2024.

[WNV+22] Laura Wartschinski, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunske. VUDENC: vulnerability detection with deep learning on a natural codebase for python. *Inf. Softw. Technol.*, 144:106809, 2022.

[WRP21] Yaza Wainakh, Moiz Rauf, and Michael Pradel. Idbench: Evaluating semantic representations of identifier names in source code. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 562–573. IEEE, 2021.

[WSL+20] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng

Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1556–1560. ACM, 2020.

[XLZ+18] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 789–799. ACM, 2018.

[YGM+22] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated classification of overfitting patches with statically extracted code features. *IEEE Trans. Software Eng.*, 48(8):2920–2938, 2022.

[YJJ14] Jongwon Yoon, Minsik Jin, and Yungbum Jung. Reducing false alarms from an industrial-strength static analyzer by SVM. In Sungdeok (Steve) Cha, Yann-Gaël Guéhéneuc, and Gihwon Kwon, editors, *21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 2: Industry, Short, and QuASoQ Papers*, pages 3–6. IEEE, 2014.

[YKH+24] Aidan ZH Yang, Sophia Kolak, Vincent J Hellendoorn, Ruben Martins, and Claire Le Goues. Revisiting Unnaturalness for Automated Program Repair in the Era of Large Language Models. *arXiv preprint arXiv:2404.15236*, 2024.

[YL20] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, pages 10799–10808. PMLR, 2020.

[YL21] Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*, pages 11941–11952. PMLR, 2021.

[YM24] He Ye and Martin Monperrus. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 10:1–10:13. ACM, 2024.

[YML+22] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 92:1–92:13. ACM, 2022.

[YMM22] He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1506–1518. IEEE, 2022.

[YWG+24] Yixin Yang, Ming Wen, Xiang Gao, Yuting Zhang, and Hailong Sun. Reducing false positives of static bug detectors through code representation learning. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*, pages 681–692. IEEE, 2024.

[YZLT17a] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 831–841, 2017.

[YZLT17b] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 831–841. ACM, 2017.

[ZSX+21] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 341–353, 2021.

[ZXK+23] Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Thanh Le-Cong, Junda He, Bach Le, and David Lo. PatchZero: Zero-shot automatic patch correctness assessment. *arXiv preprint arXiv:2303.00202*, 2023.

[ZXK+24] Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. Leveraging large language model for automatic patch correctness assessment. *IEEE Transactions on Software Engineering*, 2024.

[ZYH+23] Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. Coder Reviewer Reranking for Code Generation. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 41832–41846. PMLR, 2023.

List of Figures

1.1 User interface of our study with an example of a variable misuse bug . 4

2.1 Conceptual Overview of Neural Architectures 13
2.2 High-Level Overview over the Transformer architecture. A Transformer might learn that `if` is less relevant for detecting the variable misuse bug. 15
2.3 Joint Architecture for the Detection, Localization and Repair of Single Token Bugs. 18
2.4 Generating Artificial Training Data for Neural Bug Detection. Green operations are operations used for training neural bug detectors. The operation noop indicates that no change is required. The operation replace(t_l, r^{-1}) inverts the mutation process. 22

3.1 Code snippet taken from Defects4J/Chart#1. 26
3.2 Overview over our contextual mutation process 28
3.3 Re-weighted probability distribution $P_{\mathbf{LM}}$ for two different contexts. . . 32
3.4 Examples of all studied bug types taken from our real world benchmark. Reformatted and abbreviated for visualization. 34
3.5 Effect of the mutator on the reproducibility of real world bugs. 38
3.6 Precision and Recall for DeepBugs trained on random bugs (grey dotted), bugs introduced by SemSeed (red dashed) or contextual mutants (blue). 43

4.1 Overview over our mining process 53
4.2 Similarity of NoSStuBs with SStuBs given as the Jaccard distance between edit operations. The x-axis corresponds to the binned distance to the most similar SStuB bug and the y-axis corresponds to observed frequencies. 59

5.1 Overview of the training process 68
5.2 Effect of real bug fixes at different dataset scales on the performance on the validation set. The x-axis is the percentage of real bug fixes used during fine-tuning. 74

5.3 Effect of mutation frequency during training on the performance of Transformer-based neural bug detector on the validation set. The gray dashed line represents the average number of unique mutants that can be generated per code snippet. 76
5.4 Effect of mutation frequency during training on the performance of the graph-based neural bug detectors on the validation set. The gray dashed line represents the average number of unique mutants that can be generated per code snippet. 77
5.5 Effect of using a more realistic mutator on the performance of the Transformer-based neural bug detector on the validation set. 77
5.6 Comparison of CuBERT and our neural bug detector on Real Python projects with variable misuse bugs. 80
5.7 Comparison of CuBERT and our neural bug detector on Real Python projects with binary operator bugs. 81
5.8 Inversion test on PyPIBugs for reproducing real bugs via mutation. The x-axis is the number of mutants generated and the y-axis is the percentage of real bugs reproduced within k mutations. For the traditional random mutation, we report worst-case performance (i.e. the original bug is always sampled last) and for the improved contextual mutator we rank the mutation according to the likelihood of generating them. . . . 84

6.1 Overview of the validation strategy 95
6.2 A selective infilling problem based on our example program T and the faulty patch D. The LLM has to select between the changed code before (A.) and after (B.) applying the patch. Extra context (gray code) can be provided. 97
6.3 Example context taken from Celery for our example of a false alarm generated by a neural bug detector. The example is slightly adapted to fit the figure. 98
6.4 Comparison with Thresholding 104
6.5 Comparison to Test-based Patch Validation at $\tau = 0$ 105
6.6 Impact of validation on the ability of the neural bug detector to detect novel bugs in Python projects. 108
6.7 Impact of post-hoc validation strategies on the ability of the neural bug detector to detect novel bugs in Python projects. 109
6.8 Impact of file-level context on our example false alarm. The validator correctly rejects (with a score of -0.7) the false alarm after the file-level context is available. 110

A.1 Effect of mutation and generation strategy on the ability to reproduce real bugs on our validation set. 129

List of Tables

2.1 Examples of Software Bugs found in Python projects [AJFB21] 12

3.1 Syntactic roles distinguished by the syntactic tagger. Colors are used for visualizing the roles of tokens throughout the paper. 33
3.2 Overview for all evaluation tasks, datasets and bug types individually for each research question. For brevity, we omit the dataset statistics for individual mutator types in the same bug category, since this is shared. 35
3.3 Examples for real world bugs compared with bugs produced by our contextual mutation operator. The first three are examples for a successful reproduction of the real bug. This is followed by three cases where the mutation operator fails (real bug in comments). 40
3.4 Results on the Java real world benchmark for detection and repair (Best results marked in bold). 41
3.5 Results on the real world benchmark for detection and repair (Best results marked in bold). 42

4.1 SStub pattern statistics for CTSSB-0.9M, CSSB-2.3M, and PySStuBs . 57

5.1 Impact of training with real bug fixes on the performance of neural bug detectors on our tasks of real bug detection and correct code identification 75
5.2 Evaluation results for the improved neural bug detectors on our benchmark tasks. 78
5.3 Comparison with PyBugLab on the PyPIBugs benchmark. 79
5.4 Example of a Software Bug only found after training on Real Bugs. Code is reformatted to fit the figure. 82
5.5 Example of a Software Bug that is not found by any neural bug detector. Code is reformatted to fit the figure. 82

6.1 Example of a false alarm raised for projects in DyPyBench [BKP24] . . 93
6.2 Comparison between different LLMs as Validators at $\tau = 0$ 103
6.3 Impact of Different Context Types on Validator at $\tau = 0$ 106

6.4 Evaluation results for the validated neural bug detectors on our benchmark tasks. 107
6.5 Example of Potential Software Bug found in DyPyBench [BKP24]. Code is reformatted to fit the figure. 111
6.6 Example of Confirmed Software Bug found in `run-llama/llama_index`. Code is reformatted to fit the figure. 111
6.7 Example of Unconfirmed False Alarm found in DyPyBench [BKP24]. Code is reformatted to fit the figure. 112
6.8 Example of a Confirmed False Alarm that is still accepted by the validators found in DyPyBench [BKP24]. Code is reformatted to fit the figure. 113

A.1 Example of a software bug that cannot be reproduced via traditional mutation. Code is reformatted to fit the figure. 130
A.2 List of all Open Source Projects scanned by our neural bug detector. . . 131

Erklärung

Hiermit versichere ich an Eides statt, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Ort, Datum	Unterschrift