Metamorphic Testing for LLM-based Code Repair

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Milan de Koning

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

JetBrains
Gelrestraat 16
Amsterdam, the Netherlands
www.jetbrains.com

Metamorphic Testing for LLM-based Code Repair

Abstract

Effective LLM-based automated program repair (APR) methods can lead to massive cost reductions and have improved significantly in recent times. However, the validity of many APR evaluations as they are conducted at this point is at risk due to data leakage: Prior research has shown that LLMs can memorize solutions to problems if the evaluation benchmark overlaps with the training set, leading to overinflated results.

In this study, we examine the potential of using metamorphic transformations to mitigate the effects of data leakage. For this, we create a variant benchmark for two popular, well-established benchmarks Defects4J and GitBug-Java, and evaluate the APR performance of several LLMs on these benchmarks and their transformed counterparts. In addition, we investigate to what extent our results align with data leakage metrics from other studies.

Our results show that state-of-the-art LLMs for code repair exhibit significant performance degradation (Up to 4.1% for Claude-3.7-Sonnet) on a metamorphically transformed Defecsts4J benchmark. Moreover, we find a significant correlation between our results and the *negative log-likelihood* as a metric of data leakage. Our results demonstrate the potential of using metamorphic transformations to mitigate the overinflation of evaluation results due to data leakage. We recommend that researchers report results on both original and metamorphically transformed benchmarks in future evaluations.

Thesis Committee:

Chair:	Dr. A. Panichella, Faculty EEMCS, TU Delft
Company supervisor:	Dr. P. Derakhshanfar, JetBrains Research
External Committee Member:	Dr. S. Verwer, Faculty EEMCS, TU Delft
Committee Member:	Dr. M. Olsthoorn, Faculty EEMCS, TU Delft

Preface

This thesis marks the end of my five-year study period at TU Delft. During this time, many people have supported and helped me reach the point where I am today. First of all, I am incredibly grateful to Annibale and Pouria for their continued support and guidance. Without their insights and feedback, I wouldn't have been able to overcome the challenges that I faced during this project. Second, I want to thank everyone involved in organizing the AI4SE collaboration between TU Delft and JetBrains for creating this opportunity and making this project possible. I also want to thank Mitchell for his useful insights and for being part of my thesis committee, and Sicco for being part of my thesis committee as well.

Next, I also wish to thank the people close to me. First, my parents have provided me with the opportunities to get to this point and supported me during all of my time studying. I also want to thank my friends, who helped me take my mind off my studies and keep a balance between working and enjoying other things. Finally, I want to thank my girlfriend Milou, who was there for me during the difficult moments and supported me unconditionally.

Thank you!

Milan de Koning
June 23, 2025

Contents

List of Figures

Chapter 1

Introduction

Debugging is a widespread and time-consuming activity when building software systems. Software engineers report that they spend about 20-60% of their active work time debugging [1]. Even with the massive amount of time spent debugging, operational failures cost an estimated $1.56 trillion in the US in 2020 alone [2]. Saving even a small fraction of time and costs with automated debugging could result in massive cost reductions overall. For this reason, many studies proposing **Automated Program Repair** (APR) methods have been conducted before [3]. Several approaches for APR exist, but recently, **Large Language Model** (LLM)-based tools have shown remarkable performance on many coding tasks, including APR [4]–[8].

Extensive research effort focused on LLM-based program repair tools has aimed to improve accuracy on well-established benchmarks [5], [6], [9], [10], such as Defects4J [11]. Although it is clear that the accuracy of these tools is improving [5]–[7], several other model qualities are not reported, such as fairness [12], security [13], and in particular, robustness [14], [15]. Moreover, the validity of evaluations of LLMs on these well-established benchmarks is also questionable due to the phenomenon of **data leakage**. Data leakage occurs when an evaluation dataset overlaps with the pre-training dataset of a model and can cause optimistically biased results [16], introducing a threat to the validity of the study [17]. Ramos et al. [18] show that common benchmarks for APR, Defects4J [11] and GitBug-java [19], have most likely occurred in the pre-training datasets for several open-source LLMs. This means that the results of many APR tools tested on these benchmarks are optimistically biased and do not meaningfully reflect the true performance of such a system in a real-world scenario. This can lead to a false sense of progress and compromise the improvement of APR techniques.

To address data leakage, several new datasets have been collected with data published after the knowledge cutoff of these models [20]–[22]. However, LLMs are frequently updated with later knowledge cutoffs, and new LLMs are also introduced at a rapid pace. The new datasets collected by these papers could be in the next iteration of any popular model and suffer from the same issues that well-established evaluation benchmarks do. Even if a benchmark is not directly included in the pre-training set, evaluations may still suffer from data leakage because LLMs are trained on multiple corpora. For example, bugs, fixes, and test cases from the benchmarks may also appear in blog posts and research papers, which

```java
public int sum(int[] arr) {
  int total = 0;
  for (int i = 0; i < arr.length;
       i++) {
    total += arr[i]
  }
  return total;
}
```

(a) Code snippet

```java
public int sum(int[] arr) {
  int result = 0;
  for (int i = 0; i < arr.length;
       i++) {
    result += arr[i]
  }
  return result;
}
```

(b) Transformed code snippet

Figure 1.1: An example of a metamorphic transformation for a code snippet. The identifier `total` is changed to `result`, which does not change program semantics.

may also be part of the pre-training dataset. This makes addressing data leakage problems particularly difficult. Moreover, the creation of a new dataset requires intensive manual effort. Therefore, repeatedly and frequently creating new evaluation benchmarks is not sustainable.

Sallou et al. [17] suggest that applying **metamorphic transformations** to existing datasets could effectively mitigate the effects of data leakage. In the context of coding tasks for LLMs, metamorphic transformations are changes to source code that change the syntactic appearance of the source code without changing the behavior [14]. An example of a metamorphic transformation is shown in fig. 1.1. Applying a variety of metamorphic transformations could lead to an evaluation being less affected by data leakage because it cannot remember the solution to the transformed code bug. Despite this potential, the effectiveness of metamorphic testing as a method for uncovering and mitigating data leakage has remained unexplored until now. We hypothesize that metamorphic testing can reveal and mitigate the effects of data leakage. To the best of our knowledge, we are the first to systematically investigate this.

We aim to fill this research gap by comparing the performance of LLMs on original benchmarks and metamorphically transformed benchmarks. Our method can be summarized as follows:

1. We apply metamorphic testing to alter the syntax of code bugs without altering semantics, i.e., changing a for loop to a while loop or renaming identifiers with meaningful synonyms. All of our transformations were carefully selected to preserve the naturalness of the code snippets.

2. With these natural transformations, we transform all single-function bugs in the Defects4J and GitBug-java datasets.

3. We assess the performance of ChatGPT-4o [23], ChatGPT-4o-mini [24], Claude-3.7-Sonnet [25], Llama 3.1 8B [26], Gemma 2 27B [27], Mistral 7B v0.3 [28], and StarCoder 2 7B [29] under metamorphic testing.

We demonstrate that LLMs can perform significantly worse on the transformed benchmarks (Up to 4.1% worse on average for Claude-3.7-Sonnet). This suggests that their performance on the original benchmarks is inflated due to data leakage. We further reinforce our hypothesis by showing a correlation between a drop in performance with metamorphic testing and another estimate of data leakage: negative log-likelihood [18]. This metric indicates how 'natural' a piece of code appears to the model, providing insights into possible memorization. Our main contributions can be summarized as follows:

- We propose **CodeCocoon**, an extensively verified and tested tool for applying metamorphic transformations to source code snippets.
- We create an experimental pipeline to automatically compare the APR performance for any LLM under metamorphic testing.
- We perform an extensive empirical study, metamorphically transforming 2 benchmarks and assessing performance over these benchmarks with 7 models.
- We demonstrate the potential of using metamorphic testing for revealing and mitigating the effects of data leakage.

Chapter 2 summarizes the most relevant related works and highlights the main gaps in the research field. We provide a detailed description of our metamorphic transformation tool CodeCocoon in chapter 3. Chapter 4 explains the methods we used for our evaluations and chapter 5 describes our findings. We investigate the effects of metamorphic testing on LLM behavior for one bug in a thorough case study in chapter 6. We put our findings in a broader context and discuss threats to validity in chapter 7. Finally, we explain the main conclusions in chapter 8.

Chapter 2

Background

2.1 Automated Program Repair

Automated Program Repair is the process of automatically generating patches to software bugs [30]. There are many techniques for APR, but Huang et al. [3] define a three-step framework that APR processes typically fit into:

1. Fault Localization

This step aims to find and narrow down the defects in the code so that the APR tool can fix the appropriate piece of code. Many different Fault Localization (FL) tools have been developed [31], and APR tools usually take an off-the-shelf FL technique to locate potentially problematic pieces of code. Accurate FL is crucial for APR, as it helps APR tools generate better patches. However, FL is usually considered a separate research field from APR. Many papers introducing new APR tools assume perfect fault localization and focus only on generating and validating patches [3].

2. Patch Generation

Given the specific location of the defect, APR tools use several techniques to generate candidate patches, such as search-based [32]–[34], constraint-based [35]–[37], template-based [38]–[40], and learning-based [41]–[43] techniques. These are explained in detail in section 2.1.1. Usually, multiple candidate patches are generated per defect in this step. A significant number of patches can be generated per bug. For example, SRepair [5] generates 500 patches per bug.

3. Patch Validation

Given the candidate patches generated in the previous step, the patch validation step aims to select the best candidates from a set of patches.

Patches can be generated and validated with different objectives, but the most popular approach to APR is known as *test-suite-based program repair*. In test-suite-based program repair, the test suite is used as a specification of the desired behavior. To validate a patch, it is tested against the given test suite [5], [6], [30].

With test-suite-based program repair, patches can be divided into four categories:

- **Uncompilable patches** are invalid and cannot be compiled.
- **Failing patches** can be compiled but result in failing tests, indicating incorrect behavior.
- **Plausible patches** pass the tests.
- **Correct patches** pass the tests and are manually verified to be semantically identical to the human fix. This check is done because test suites cannot always exhaustively test for every possible problem and may pass incorrect patches. Correct patches are always a subset of the plausible patches. However, it is not always feasible to manually check the correctness of all plausible patches.

2.1.1 APR techniques

APR techniques can be divided as follows [3]:

- **Search-based:** This approach was the first repair technique to be explored and has shown many possibilities for program repair. It involves searching a predefined search space with the help of a metaheuristic. For example, Genetic algorithms are commonly used. This involves using genetic operators such as mutation and crossover to generate and evolve candidate patches. The fitness of candidate patches can be calculated by counting the number of failing tests. This has proven relatively effective. However, this technique is limited by the massive search space and very high computational costs [3], [32]–[34].
- **Constraint-based:** This approach is much more efficient than search-based APR because the problem is formulated as a set of formal constraints and can be solved with a state-of-the-art constraint solver. The downside of this technique is that it relies heavily on correct constraints, making it less flexible and requiring a lot of costly manual effort to formulate [3], [35]–[37].
- **Template-based:** To limit the search space, template-based APR methods define a set of bug-fix templates that can be applied to a buggy piece of code to generate candidate patches. This approach is very efficient for repairing specific types of bugs, but is limited by the templates and cannot solve defects that are not summarized by its templates [3], [38]–[40].
- **Learning-based:** These methods apply deep learning to gain high-level repair knowledge. Learning-based APR tools have shown many promising results and the potential for deep learning for APR. [3], [41]–[43] These models are sometimes referred to as neural program repair(NPR). Within learning-based APR, LLM-based APR has demonstrated significantly better performance than other learning-based APR tools [4]–[7].

2.2 Large language models

Large Language Models (LLMs) are neural networks designed for token prediction. Tokens are smaller, discrete parts of a text, such as words, sub-words, characters, or symbols, depending on the technique for breaking text into tokens. LLMs are tasked with predicting the next token given a sequence of tokens. They produce a distribution over all possible tokens, with tokens more likely to follow the input sequence having a higher probability of being chosen as the next token. They are first trained on a massive text dataset in a self-supervised manner. Then, they can be fine-tuned for specific tasks. For flexibility, many LLMs are fine-tuned to follow instructions, so that they can be applied to a wide variety of tasks [44]–[47]. These include coding tasks such as code generation [48], code summarization [49], vulnerability detection [50], and APR [4]. Popular LLMs for coding tasks include ChatGPT [51], Claude [25], and DeepSeek [52] [53].

2.2.1 LLM-based APR

Many studies proposing LLM-based APR tools have been published in recent years. These methods have shown remarkable performance. The abilities of LLMs are leveraged in several distinct ways [4]:

1. **Fine-tuning** an LLM on a small, task-specific dataset. This is an intuitive way to make LLMs behave desirably. This technique was mainly employed to integrate early, smaller LLMs such as CodeT5 [54] and CodeBERT [55] into APR tools. [4], [56]–[60]

2. **Few-shot Learning** involves an LLM being asked to perform a task with a few examples. This is used for APR mainly in mid-sized LLMs, such as CodeX [61]. [4], [62]

3. **Zero-shot Learning** refers to the LLM being prompted to perform program repair without explicit examples, instead using its pre-existing knowledge and understanding of the task. This technique is popular when leveraging larger LLMs such as ChatGPT [51] in an APR tool. [4], [63], [64]

However, even though these methods have shown great performance, there are some issues. Firstly, LLMs can suffer from the effects of **data leakage**, where the LLM has seen a test benchmark dataset during pre-training, leading to biased output results. We go into more detail on the issue of data leakage in section 2.3. Secondly, the robustness of many of these tools remains unknown. As shown in [14], [15], [65]–[68], many deep code models lack robustness to metamorphic transformations. We explain this further in section 2.4.

2.3 Data leakage

Data leakage occurs when there is an overlap between the benchmark used to evaluate an LLM and the dataset used for training the model. It has been shown that this can lead

to optimistically biased evaluation results [16], [69], and does not reflect a system's true performance in a real-world scenario. This can lead to a false sense of progress, meaningless comparative evaluations and a lack of generalizability. Many researchers have called for action to mitigate the effects of data leakage during evaluations [16]–[18], [69]–[71].

Data leakage is especially problematic in APR, for several reasons: First, creating an APR benchmark requires intensive manual effort, involving identifying, reproducing, and isolating bugs from various open-source projects [11]. Because of this, there are only a few well-established benchmarks that are widely used within the APR community. Since these are so frequently mentioned and discussed in public sources, there is a high risk that these datasets are included in pre-training datasets any number of times. Furthermore, in LLM-based APR, ChatGPT [51] is the most popular family of models [4]. Both the parameters and training set for these models are closed-source, making it difficult to determine whether a benchmark overlaps with the training set.

Finally, the issue of data leakage is often overlooked in APR studies. For example, [5], [6] both achieve remarkable APR performance on the Defects4J using ChatGPT as an underlying LLM. However, they do not mention or acknowledge the concept of data leakage. Most likely, the results that they report are optimistically biased, and this means we cannot confidently apply these tools in a real-world scenario. As also mentioned by Zhang et al. [4] in their literature review on LLMs for code repair, we urgently require techniques for mitigating data leakage.

Nevertheless, several methods exist for data leakage detection, and there have been efforts to mitigate data leakage as well.

2.3.1 Data leakage detection

Several studies have demonstrated data leakage in LLMs, for both natural language samples and code samples.

Al-Kaswan et al. [72] show that it is possible to extract a part of LLMs training samples, because the models could memorize and reproduce the exact samples. Building on previous research, they show that not only natural language samples but also code samples can be memorized by LLMs. Furthermore, they show that models with a higher parameter count are more prone to memorization and reproduction of training samples.

Furthermore, Xu et al. and Li [73], [74] Use *perplexity* and *n-gram accuracy* to estimate levels of data leakage in LLMs. *Perplexity* is a measure of the uncertainty of a model when predicting the next token in a sequence. It is based on the *negative log-likelihood* (NLL) of the model over a sequence. The NLL is defined as follows:

$$NLL(x_i) = -\log p_\theta(x_i|x_{<i})$$

Where p_θ is the probability distribution over possible next tokens produced by the model, x_i is the current token, and $x < i$ is the sequence that came before token i. *Perplexity* is defined as the exponentiated average NLL:

$$PPL(X) = \exp(\frac{1}{|X|}\sum_{i=0}^{|X|} NLL(x_i))$$

This metric gives insight into the model's confidence when predicting tokens in a sequence. If the perplexity over a sequence is very low, it appears natural to the model and was most likely memorized. Furthermore, n-gram accuracy measures how well a model can exactly reproduce sequences. Xu et al. [73] define the n-gram accuracy over a dataset as follows:

$$\text{N-gram accuracy}(X) = \sum_{i=0}^{S} \sum_{j=0}^{K} I(X_{start_j;start_j+n}, \hat{X}_{start_j;start_j+n}))$$

Where the size of the dataset is S, $start_j$ is the index of the j-th starting point, I is the exact match indicator function, $X_{start_j;start_j+n}$ is the n-gram of the actual sample, and $\hat{X}_{start_j;start_j+n}$ is the n-gram predicted by the model. An n-gram is a sequence of n tokens. Contrary to the *perplexity* metric, a very high n-gram accuracy indicates that the model can (almost) exactly reproduce a sequence. This is also a sign of data leakage. Together, *perplexity* and *n-gram accuracy* can be used to compare a model's output for suspected leaked snippets and snippets that were not leaked. Using these metrics, Xu et al. and Li [73], [74] show evidence of significant memorization.

Finally, Ramos et al. apply a similar method, computing the NLL and 5-gram accuracy for popular coding models over popular Java benchmarks, such as Defects4J [11] and GitBug-Java [19]. They compare the metrics over popular Java benchmarks to a manually created set of Java repositories after the knowledge cutoff of these models. Their results reveal that several models have a significantly lower NLL on the Defects4J dataset than on newer Java repositories, indicating the presence of data leakage. The 5-gram accuracy does not show as strong a pattern. Overall, their results highlight the risk of using older, well-established benchmarks for evaluation, due to data leakage.

2.3.2 Data leakage mitigation

To mitigate the effects of data leakage, several studies have created new benchmarks to evaluate APR model performance [20]–[22]. This is also the recommendation made in the survey on LLMs for code repair by Zhang et al.[75]. However, this is not a long-term solution, because these new benchmarks are published, and LLM pre-training datasets are often scraped from public sources. Due to the rapid rate at which new LLM versions are released, these new benchmarks can quickly become obsolete.

Bradbury and More [76] suggest a technique for addressing data leakage, which involves dynamically instantiating concrete benchmark samples from templates. With this approach, they create several variants of the HumanEval benchmark [61], a collection of coding problem descriptions in natural language. This benchmark is commonly used to evaluate the coding skills of LLMs. In their study, they show that popular LLMs such as ChatGPT [51], Claude [25], and [26] perform worse on their variants of the HumanEval dataset compared to the original dataset. They demonstrate the effectiveness of their approach in creating benchmark variants that are more robust to data leakage. However, their technique also requires an extensive amount of manual effort to craft benchmark templates.

As also suggested in [17], benchmarks could be metamorphically transformed to combat the effects of data leakage.

2.4 Metamorphic testing

Metamorphic testing is a technique commonly used to evaluate the robustness of models for code-related tasks. It involves transforming the syntax or Abstract Syntax Tree (AST) of input code snippets without changing the semantic behavior of the code snippet. By observing changes in output between the original and transformed snippets, we can evaluate whether the model under test is robust to these transformations. [14], [15]

This also holds for the APR task. If an APR tool can generate a valid patch for a buggy code snippet, the tool should also be able to generate a valid patch if this code snippet is transformed in a way that does not alter the semantics of the code snippet. An example of a metamorphic transformation to a code snippet is shown in 1.1. The code snippet has a different syntax, but will still behave the same. An APR tool should still be able to fix a bug in the transformed snippet if it was able to fix a bug in the original code snippet.

As shown in [14], [15], [65]–[68], many models for code-related tasks are vulnerable to these kinds of transformations and can change their outputs based on irrelevantly transformed input. Many of these transformations exist, including not only identifier renaming, but also structural transformations, such as changing a `for` loop to a `while` loop, or reversing the blocks of an `if` statement.

Several works argue that these transformations should result in code snippets that appear as natural as possible to evaluate the models under a realistic setting [65], [77], [78]. Applying unnatural transformations can lead to false alarms, where the model would not fail in a real-world scenario. In this study, we apply several natural transformations, such as identifier renaming with synonyms and several structural transformations. Chapter 3 explains the transformations in detail.

2.5 Related work

Many studies have applied metamorphic testing to deep code models previously. We provide a condensed overview of the studies targeting APR in particular here.

Ge et al. [79] investigate the robustness of four SOTA-at-the-time APR models with metamorphic testing. These models (Recoder [80], CoCoNut [81], SequenceR [42] and Tufano [41]) are deep neural networks, but they are not LLMs. They find that even the most robust model, SequenceR, cannot fix all transformed inputs in 20% of cases where it was able to fix the original bug. Four different transformations are applied in their study. They also analyze how perfect fault localization impacts robustness and find that better fault localization leads to better robustness. If the fault localization is incorrect, the models suffer even more from the transformations.

Le-cong et al. [78] also perform metamorphic testing for NPR models (SequenceR [42], Recoder [80], RewardRepair [82], SelfAPR [83], and AlphaRepair [64]), but focus more on the naturalness of transformations. They conduct studies with experts to quantify the naturalness of code snippets and find that 25% of prediction changes are caused by unnatural code snippets. They argue that we should not waste efforts trying to rectify these false alarms, as these are unlikely to occur in a real-world scenario. Furthermore, they make some first steps towards automatic naturalness assessments using early LLMs, such

as GPTNeo [84], BLOOM [85], and CodeLlama [86]. They use the cross-entropy as a measure of how ”surprised” the LLM is by the code snippet to assess its naturalness.

Moreover, Li et al. [87] perform a small study with top-performing LLMs for code generation and test how well they generalize when prompted to do code repair on Defects4J and a transformed version of Defects4J. They show that there is a drop in performance, but they attribute this to a lack of robustness only and do not consider the concept of data leakage in their explanation.

Finally, Xue et al. [88] extend beyond previous works investigating traditional NPR models and applying metamorphic testing to LLM-based APR. They apply it to several popular open-source LLMs such as LLama 3, Mistral, and CodeGemma. Even though these models have shown better APR performance than previously investigated models, they are still not comparable to the best-performing LLMs for APR [89]. Furthermore, they substantially lift the robustness of the LLMs by training a model to revert the transformations and then feeding this output into the LLM. However, the generalizability of this approach remains questionable, as they do not test this approach under unseen transformations.

Due to recent LLM developments, the models under test in these papers have become outdated. We aim to address this gap by evaluating the current SOTA LLMs for APR [89]. We also aim to investigate whether prediction changes are due to data leakage. The possibility of data leakage issues is not discussed in the aforementioned works, while it could have significantly impacted the observed results.

Even though there are other methods of detecting data leakage, our proposed method of evaluating data leakage through metamorphic testing comes with unique benefits. First of all, our method can be applied to individual code snippets to gauge whether a snippet is leaked. NLL and n-gram accuracy are greatly impacted by the naturalness of a code snippet, and can only be compared to other code snippets. Furthermore, our method does not require extra information beyond the model output tokens, such as negative log-likelihood, which may not be available for closed-source models. Finally, our proposed method is not only applicable for data leakage detection, but our findings suggest that it can be applied to mitigate the effects of data leakage.

Chapter 3

CodeCocoon

We have developed CodeCocoon to conduct metamorphic testing for code models. It can transform Java code snippets with several different types of transformations, such as structural and renaming transformations, while making sure that the results remain natural to a human judge. Although several alternatives exist, CodeCocoon is, to the best of our knowledge, the only tool that applies natural transformations and is extensively tested and verified to preserve code behavior. We focus on APR in this paper, but CodeCocoon can be applied to do metamorphic testing for any code-related task. CodeCocoon is available at `https://github.com/milandekoning/CodeCocoon`.

3.1 System Overview

CodeCocoon consists of several components, as shown in fig. 3.1. The user provides the dataset and specifies the transformations and their order in a configuration to the main driver class (1). This configuration is passed to the Transformer factory (2) to create a composite transformer (3) that satisfies this configuration. Each code snippet in the dataset is passed to the composite transformer, which applies the transformers (4) specified in the configuration. A dedicated transformer is implemented for each transformation. To create natural alternatives for identifier renaming transformations, we implemented a synonym generator (5) that produces a natural synonym for an identifier given the code context. We prompt an LLM (6) to generate alternative identifiers. The system keeps track of the number of transformations that were applied and how the identifiers were renamed.

3.2 Transformations

We implement several different structural and renaming transformations. We only use natural transformations, as unnatural transformations evaluate the model under non-realistic input and can lead to false alarms [77], [78]. The transformations we implemented are a subset of the transformations in [78], where a human evaluation was done to determine the naturalness of several transformations. Due to time constraints, we opted to implement a

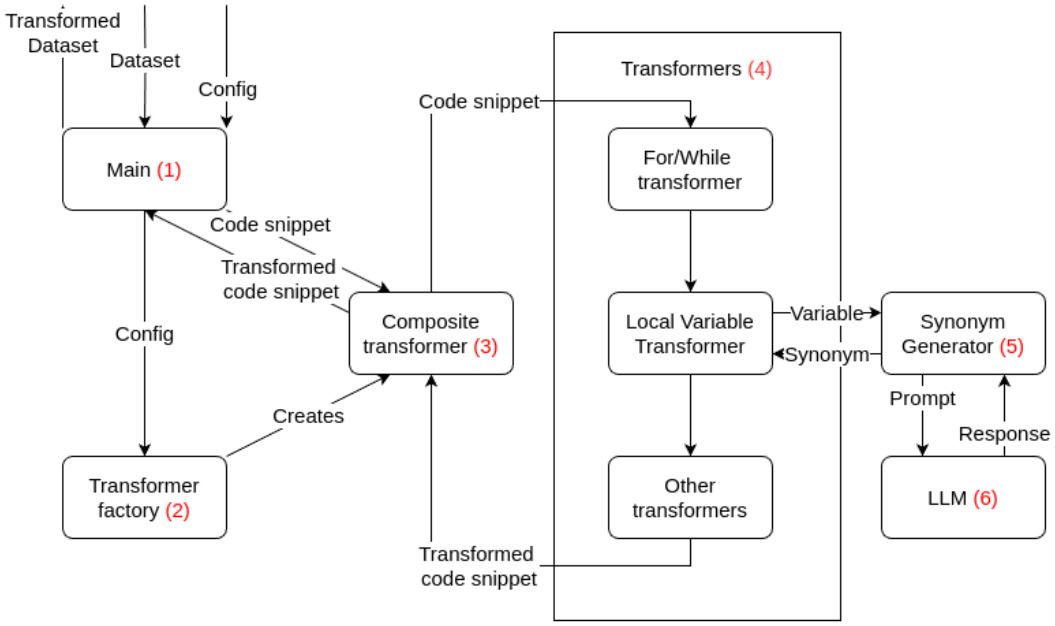

Figure 3.1: System overview for CodeCocoon. A dataset and a transformation configuration are supplied to the driver class (1). The transformer factory (2) creates a composite transformer (3) that satisfies the configuration. Then, each snippet in the dataset is fed through the transformers (4). In this example, the for-to-while transformer is applied first, then variables are renamed with a synonym generator (5), which uses an LLM (6), and other transformers are applied. Finally, the composite transformer returns the transformed snippet. When all snippets are transformed, the full transformed dataset is returned.

subset of the transformations that were judged relatively simple to implement and apply to prevalent code structures. The chosen transformations are shown and defined below.

3.2.1 Renaming transformations

We replace identifiers in code snippets with contextual synonyms to preserve naturalness. The synonyms are generated by asking ChatGPT-4o-mini [24] to give a synonym for a variable given the context (the rest of the code snippet). We have three such transformations: Function renaming, Parameter renaming, and Local variable renaming. They are described in detail below.

1. Rename function replaces the name of a function with a context synonym. We perform a collision check to make sure that the new function name is not already used in the same method. If the function is recursive, all of the recursive calls are also updated. Moreover, we choose not to rename some functions. Even though we use synonyms, some names, such as `equals`, `toString`, etc., carry inherent meaning that is not present in a synonym such as `isEqual`, because these functions can be implicitly called without using the actual

```java
public int add(int a, int b) {
  return a + b;
}
```

(a) Original Code snippet

```java
public int sum(int a, int b) {
  return a + b;
}
```

(b) Transformed code snippet

Figure 3.2: An example of the function renaming transformation.

```java
public int add(int a, int b) {
  return a + b;
}
```

(a) Original Code snippet

```java
public int add(int x1, int x2) {
  return x1 + x2;
}
```

(b) Transformed code snippet

Figure 3.3: An example of the parameter renaming transformation.

```java
public int sum(int[] arr) {
  int total = 0;
  for (int i = 0; i < arr.length;
       i++) {
    total += arr[i]
  }
  return total;
}
```

(a) Code snippet

```java
public int sum(int[] arr) {
  int result = 0;
  for (int i = 0; i < arr.length;
       i++) {
    result += arr[i]
  }
  return result;
}
```

(b) Transformed code snippet

Figure 3.4: An example of the local variable renaming transformation.

name. Therefore, we do not change the name of the function if it is inherited from the base `Object` Java class. An example of this transformation is shown in fig. 3.2.

2. Rename parameter replaces the name of a parameter with a context synonym. Similarly to the function renaming transformation, we make sure that there are no collisions. An example is shown in fig. 3.3.

3. Rename local variable replaces the name of a local variable with a context synonym. Similarly to the previously mentioned transformations, we make sure that there are no collisions. An example is shown in fig. 3.4.

3.2.2 Structural transformations

We implemented several structural transformations. These transformations result in a differently structured AST, while preserving identical behavior. They are explained in more detail below.

1. For to while replaces a for loop with a while loop. We prepend the initialization (`int i = 0`) to the loop. Because this lifts the initialization of the loop variable to a higher

```java
public int sum(int[] arr) {
  int total = 0;
  for (int i = 0; i < arr.length;
      i++) {
    total += arr[i]
  }
  return total;

}
```

(a) Original Code snippet

```java
public int sum(int[] arr) {
  int total = 0;
  int i = 0;
  while (i < arr.length) {
    total += arr[i]
    i++;
  }
  return total;
}
```

(b) Transformed code snippet

Figure 3.5: An example of the for loop to while loop transformation.

```java
public int test(int a) {
  if (a == 0) {
    return a;
  } else if (a == 1) {
    return 10;
  }
  return 0;


}
```

(a) Code snippet

```java
public int test(int[] arr) {
  if (a == 0) {
    return a;
  } else {
    if (a == 1) {
      return 10;
    }
  }
  return 0;
}
```

(b) Transformed code snippet

Figure 3.6: An example of the nest else-if transformation.

scope, we only apply this transformation if the loop variable is not defined elsewhere in the same scope. Furthermore, the update (`i++`) is appended at the end of the loop. Because a `continue` statement calls the update instruction in a for loop, but not in a while loop, we must also add the update instruction before any `continue`. An example of this transformation is shown in fig. 3.5.

2. Nest else if replaces an else-if block with an if condition nested inside an else block. An example is shown in fig. 3.6.

3. Reverse if negates the condition of an if-else statement and swaps the then and else blocks. Example is shown in fig. 3.7.

3.2.3 Expression transformations

We opted to add expression-level transformations, where expressions are rewritten to equivalent forms. They are described below.

1. Swap equals operands swaps the operands of an == expression. This is only allowed if the child expressions do not contain any assignments, because they are evaluated from left

```
public boolean isPositive(int a)
    {
  if (a > 0) {
    return true;
  } else {
    return false;
  }
}
```

(a) Code snippet

```
public boolean isPositive(int a)
    {
  if (a <= 0) {
    return false;
  } else {
    return true;
  }
}
```

(b) Transformed code snippet

Figure 3.7: An example of the reverse if transformation.

```
public boolean isEqual(int a, int
    b) {
  return a == b;
}
```

(a) Original Code snippet

```
public boolean isEqual(int a, int
    b) {
  return b == a;
}
```

(b) Transformed code snippet

Figure 3.8: An example of the swap equals operands transformation.

.

```
public boolean greater(int a, int
    b) {
  return a > b;
}
```

(a) Original Code snippet

```
public boolean greater(int a, int
    b) {
  return b < a;
}
```

(b) Transformed code snippet

Figure 3.9: An example of the swap equals operands transformation.

to right, and changing the execution order also changes the behavior. An example is shown in fig. 3.8.

2. Swap relation operands swaps the operands of a relational expression and updates the expression accordingly. These relational expressions include (>, >=, <, <=). This is only allowed if the child expressions do not contain any assignment operations, because they are evaluated from left to right, and changing the execution order also changes the behavior. An example is shown in fig. 3.9.

3. Expand Unary Increment expands a unary increment or decrement operation into an addition/subtraction by 1. An example is shown in fig. 3.10.

All transformations are applied deterministically; If a transformation can be validly applied, it is applied by the system.

```java
public int addOne(int a) {
  a++;
  return a;
}
```

(a) Original Code snippet

```java
public int addOne(int a) {
  a += 1;
  return a;
}
```

(b) Transformed code snippet

Figure 3.10: An example of the expand unary increment or decrement transformation.

3.3 Implementation

CodeCocoon is implemented in Java entirely. The transformers are implemented with `JavaParser` [90]. We use `JavaParser` to parse the snippet into a model of the AST. Then, CodeCocoon uses the visitor pattern [90] to apply the transformations. The visitor pattern entails that every transformer is a child of the `ModifierVisitor` base class, which has functionality for performing an AST traversal. The transformers override the traversal function for the AST element to transform and apply the required changes before continuing the traversal. When the AST traversal is complete, we convert the AST model back to a string and store it.

To ensure correctness, CodeCocoon was extensively unit-tested, with a branch coverage of over 80%. Furthermore, 30 transformed snippets were taken and manually validated to be semantically identical to the original snippets.

We access ChatGPT-4o-mini for synonym generation via the JetBrains internal AI system. To reduce costs, we store every request and response in a 'persistent cache'. The system retrieves the response from the cache if an identical request has already been made before.

3.4 Usage

The system is designed for use within an evaluation pipeline and is not designed for direct user interaction. It can be used by providing a dataset of code snippets and a transformation configuration. The dataset of code snippets is expected as a simple JSON file, where each snippet is stored as a string. An example of this is shown in fig. 3.11. The configuration contains the input and output paths, the order in which transformations should be applied, and the LLM configuration for synonym generation. Along with the transformed snippets, the system also outputs the transformations applied per snippet and the mapping from old to new identifier names for further analysis.

3.5 Related systems

Several systems that aim to achieve a similar goal already exist. Applis et al. [14], [15] propose LAMPION to apply metamorphic transformations. However, there are a few drawbacks to their tool. Firstly, it is not designed to create natural code transformations, which is a requirement for our study. Secondly, it is based on `spoon` [91], a library for parsing code

```
{
  "Test-1": "public static void test() { ... }",
  "Test-2": "public int stop() { ... }",
  ...
  "Run-7": "public void run() { ... }"
}
```

Figure 3.11: Expected dataset format for CodeCocoon . Each code snippet has a key to allow linking the original and the transformed code snippet after applying the transformations.

and editing the AST. However, it does not support every language construct that occurs in our dataset. Because of this, we judged that `JavaParser` [90] would be more appropriate to use. Ge et al. [79] and Rabin et al.[92] propose RobustNPR and ProgramTransformer, two other tools for applying metamorphic transformations. However, neither is designed for natural transformations, but neither has been extensively tested and verified. As mentioned in section 3.2, some transformations require extra checks or functionality to preserve semantic behavior. We implement a new tool because the others do not produce natural code snippets and lack extensive verification.

Chapter 4

Methodology

To the best of our knowledge, we are the first to evaluate data leakage through metamorphic testing. We ask the following research questions:

- **RQ1:** *To what extent does metamorphic testing affect the performance of SOTA LLMs for code repair?*
- **RQ2:** *To what extent do specific metamorphic transformations affect the performance of SOTA LLMs for code repair?*
- **RQ3:** *To what extent can an APR performance decrease under metamorphic testing be attributed to data leakage?*

To answer these research questions, we evaluate APR model performance under metamorphic transformations using an experimental pipeline. We address RQ1 by comparing repair success rates before and after applying the transformations. For RQ2, we analyze the correlation between the transformations applied and the drop in APR performance. We address RQ3 by testing a correlation between APR performance drops and other estimations of data leakage by Ramos et al. [18].

4.1 APR setting

In our experiments, we apply APR by prompting the model under test similarly to the method SRepair proposed by Xiang et al. [5]. We use Chain-of-Thought [93] techniques to prompt our APR model to analyze the root cause of the bug, suggest solutions, and implement these solutions. In our prompt, we provide the buggy function, Javadoc, a random trigger test case, and the stack trace of this trigger test. An example of this is shown in fig. 4.1. We extract the patches from the response and replace the buggy function with the generated patches in the project before executing the tests.

The key difference between our implementation and the implementation of SRepair [5] is that we let the LLM simultaneously reason about the root cause and possible solutions and implement these solutions to save time and costs. This setting for code repair is sufficient to demonstrate the robustness and data leakage problems present in LLMs.

Finally, we apply code repair under the assumption of function-level fault localization, where the model is only given the function and no more information about where the bug is located. This is in contrast to other studies [78], [79], where code repair is evaluated under perfect, line-level fault localization. In real-world scenarios, function-level fault localization requires much less effort than line-level localization, making function-level fault localization a more practical setting.

4.2 Experimental setup

Dataset

We use the Defects4J [11] and GitBug-Java [19] datasets. As explained in chapter 2, we use the Defects4J [11] dataset, because it is the most well-established bug benchmark and can be used to demonstrate the effects of data leakage [18]. We opted to conduct our experiment with the GitBug-Java benchmark as well, since it is a more recent, yet commonly used APR benchmark. In addition, because this benchmark is much more recent, there is a lower risk of data leakage on this benchmark, which leads to interesting insights when compared with the older Defects4J benchmark.

Xiang et al. [5] already extracted all relevant information for all single-function bugs in Defects4J into a convenient format. From this, we extracted the information that was required for our experiment. We extracted the relevant information from the GitBug-Java dataset ourselves.

Bug filtering

In our results, we removed the bugs that were too 'easy' or too 'hard', i.e., a success rate of 0% or 100% for both original and transformed versions of a bug, for our three state-of-the-art models. This results in a subset of 'Balanced Complexity Bugs': Bugs that are appropriately difficult. The distribution of bugs over the projects in the datasets is shown in table 4.1.

In cases where results are only considered across a single model, bugs with a success rate of 0% or 100% for only that model were filtered out. We perform this filtering to highlight meaningful results. The point of this study is not to show the absolute performance of these models, so removing the easiest and most difficult bugs is a valid way to highlight a meaningful performance difference.

Models

We evaluate three state-of-the-art models for code repair: ChatGPT-4o [23], ChatGPT-4o-mini [24] and Claude-3.7-Sonnet [25]. Furthermore, to conduct a correlation analysis with the results from [18], we must conduct a study of the behavior of several open-source models, as they were only able to include results over those models in their study. We conduct our experiments on Llama 3.1 8B [26], StarCoder 2 7B [29], Gemma 2 27B [27], and Mistral 7B v0.3 [28]. Because our method uses an instruction prompt, we took the instruction fine-tuned versions of these models from Hugging Face [94].

```
You need to first analyse the buggy code, trigger test and error message.
Then analyse the root cause and finally try to provide a repair suggestion
to fix the bug. Note that the bug can be fixed by modifying only the given
buggy code; do not attempt to modify the class, add new functions, or
conduct further testing.
1. Buggy Function:
/**
* Returns a paint for the specified value.
*
* @param value  the value (must be within the range specified by the
*         lower and upper bounds for the scale).
*
* @return A paint for the specified value.
*/
public Paint getPaint(double value) {
    double v = Math.max(value, this.lowerBound);
    v = Math.min(v, this.upperBound);
    int g = (int) ((value - this.lowerBound) / (this.upperBound
            - this.lowerBound) * 255.0);
    return new Color(g, g, g);
}
2. Trigger Test:
    public void testGetPaint() {
        ...
    }

3. Error Message:
java.lang.IllegalArgumentException: Color parameter outside of expected range:
Red Green Blue
        at org.jfree.chart.renderer.GrayPaintScale
            .getPaint(GrayPaintScale.java:128)  return new Color(g, g, g);
        at org.jfree.chart.renderer.junit.GrayPaintScaleTests
            .testGetPaint(GrayPaintScaleTests.java:107)
                c = (Color) gps.getPaint(-0.5);

First, analyze the trigger test and error message, and then analyze the root
cause of the buggy function in the format 'Root Cause: {content}'. Provide a
detailed patch suggestion for resolving this bug. Your suggestion should be in
the format 'Suggestion 1: {suggestion title}\n{full patched function}', etc.
The patched function suggestion should be surrounded with ```java\n```
Make sure to return the entire function, do not leave out parts.
```

Figure 4.1: Example of the APR prompt for the `Chart-24` bug. We supply the Javadoc and buggy function (1), A trigger test (2), and the stack trace (3). We emphasize that the model should find the root cause before suggesting solutions to elicit reasoning behaviour.

Table 4.1: Distribution of bugs over projects in Defects4J and GitBug-Java. Single-function bugs are a subset of all bugs, and balanced complexity bugs are a subset of those bugs. The Balanced Complexity bugs are the bugs that have a success rate that is not 0% or 100% for at least one model. Note that the GitBug-Java dataset does include 199 bugs spread over 55 projects. However, 44 of these projects did not include any single-function bugs, which are not shown in this table.

Dataset	Project	#Bugs	#SF bugs	#Balanced Complexity Bugs
Defects4J	Chart	25	16	11
	Cli	30	28	21
	Closure	140	105	79
	Codec	13	11	9
	Collections	2	1	0
	Compress	40	36	31
	Csv	13	12	10
	Gson	12	9	8
	JacksonCore	18	13	9
	JacksonDatabind	85	67	55
	JacksonXml	5	5	5
	Jsoup	58	53	43
	JxPath	14	10	4
	Lang	56	40	33
	Math	102	74	60
	Mockito	30	24	13
	Time	22	16	10
	Overall	665	520	401
GitBug-Java	Simple-DSL	2	1	1
	Ari-proxy	1	1	1
	Crawler-commons	1	1	1
	Jansi	1	1	1
	Java-solutions	1	1	1
	Java-stellar-sdk	3	1	1
	Jsoup	29	6	5
	Openapi-to-plantuml	1	1	0
	Semver4j	3	1	1
	Spring-retry	2	1	0
	Traccar	81	37	23
	Overall	125	52	35

We ran these models locally with vLLM [95]. We omitted the CodeGen 6B [96] model from our experiments, as there is no instruction-tuned version available on Hugging Face, and the CodeGen architecture is not supported by vLLM. Furthermore, we omitted the Llama 70B [26] model because of limited computational resources. We access the OpenAI models and Claude-3.7-Sonnet through the JetBrains internal AI system.

Technical specs
The experiments were run on a Linux 20.04 server with 512 GB RAM, 4 AMD EPYC 7H12 64-Core Processors, and an Nvidia A40 GPU.

Samples
We prompted the models 10 times for each original bug and 10 times for each transformed bug to gain a statistically sound performance estimate, as suggested by [97]. Each sample is taken with a new chat session. Note that we asked the state-of-the-art models to produce 5 patches per prompt, and asked the open-source models to only produce 1 patch, due to the context length and output limits of those models. We judge that it is fair to consider the entire prompt a success if at least one of the patch suggestions passes the tests. This is because we prompt it to generate unique solutions, even in cases where there may only be one possible solution. Furthermore, we judged that five is a realistic number of patches to test in a real-world scenario.

Metrics
We define the model performance on a given bug in terms of Success Rate (*SR*):

$$SR = \frac{\text{\#prompts that produce} \geq 1 \text{ correct patch}}{\text{\#prompts}}$$

This is the percentage of prompts that result in at least one plausible patch. This means that we consider a prompt successful if at least one of the suggested patches passes all of the tests. We consider this to be a measure that is meaningful in real-world situations. First of all, taking a large number of samples and considering the bug to be solved when at least one patch is correct, like in [5], may be sufficient for measuring research progress but is not feasible in real-world scenarios. We define the success rates on the original and transformed functions as SR_{orig} and SR_{trans}. We define the difference in success rate as $SR_{diff} = SR_{trans} - SR_{orig}$

Furthermore, because our APR models are nondeterministic, we must take multiple samples per code snippet to obtain a statistically valid performance estimate. To determine the presence of a statistically significant difference between patches for the original and transformed functions, we use the Python implementation of Fisher's exact test [98], [99]. The effect size is measured in terms of the odds ratio (*OR*):

$$OR = \frac{SR_{orig}/SR_{trans}}{FR_{orig}/FR_{trans}}$$

where

	Success	Failure
Original	SR_{orig}	FR_{orig}
Transformed	SR_{trans}	FR_{trans}

We report results as statistically significant if the p-value is smaller than 0.05, and as very statistically significant when the p-value is below 0.01.

Finally, only the test results are used to calculate performance metrics due to the extensive manual effort required for correctness checking. The difference in test results sufficiently demonstrates the effect of metamorphic testing.

Analysis of transformation impact

To analyze how different transformations and combinations of transformations impact the success rate, we use the two-way ANOVA test [100]. This test determines whether the variance in the dependent variables (SR_{diff}) and be attributed to the independent variables, which are the frequency of each transformation type (e.g., the number of variable names being changed).

If the F-statistic (that is, the ratio between explained and unexplained variance) is statistically significant (p-value$<$0.05), it indicates that one or more transformation types (or their combinations) significantly impact the LLM success rate. We limit our analysis to the interaction between at most three factors. There are multiple reasons for this choice. First, including too many interacting transformations makes the results difficult to interpret. Second, the number of possible factor combinations grows exponentially with the number of transformations, which drastically increases the complexity of the model. This leads to a combinatorial explosion in both computational cost and multiple testing risk. Finally, higher-order interaction terms require substantially more data to estimate reliably. As the number of interacting factors increases, the number of observations needed to cover all possible combinations (cells in the design matrix) grows rapidly. In our dataset, higher-order interactions would result in many underpopulated or empty combinations, leading to overfitting and unreliable estimates.

Correlation analysis

To perform correlation analysis for RQ3, we calculate the Spearman correlation [101] between the difference in success rate SR_{diff} for each bug with other bug-specific metrics. For RQ3, we test the correlation between SR_{diff} and the NLL as provided by Ramos et al. [18]. We should use the Spearman correlation because our results are not normally distributed. We use the Python implementation of the Spearman correlation [102]

The results provided by Ramos et al. represent the average NLL over the full **original** Java file containing the bug. These values give insight into how familiar a model is with a given piece of code. Their results only cover a subset of the Defects4J dataset, so we use their replication package to collect results over the complete benchmark.

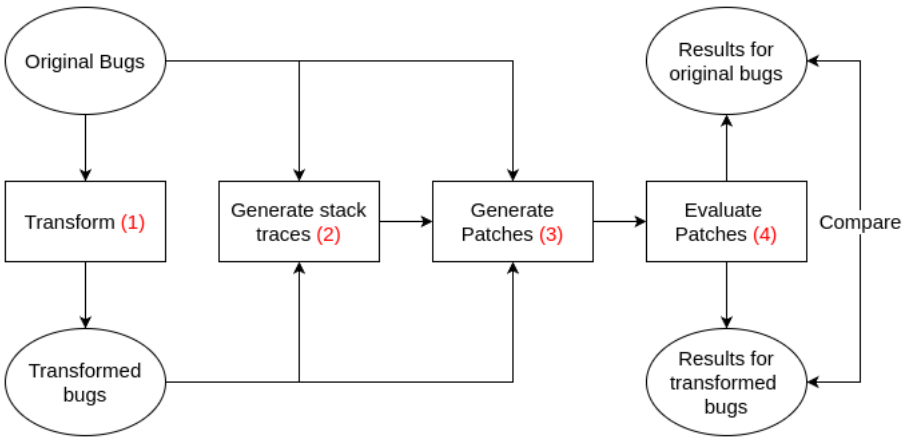

Figure 4.2: Experimental pipeline. The original dataset is transformed (1), and stack traces are generated (2) for the failing tests. Afterward, we generate patches (3) using the original bugs and transformed bugs as input for our APR model. Finally, we evaluate the quality of the patches (4) by running the tests.

4.3 Experimental pipeline

Having defined our experimental setup, this section describes the pipeline for transforming the dataset, generating patches with LLMs, and evaluating the patches. Our code and results are available at `https://doi.org/10.5281/zenodo.15719286` An overview of the experimental pipeline is shown in fig. 4.2 and consists of the following stages:

1. Transform

This stage uses CodeCocoon (explained in chapter 3) to transform all bugs in the dataset. The Javadoc and trigger test cases are updated with regular expressions to reflect the parameter name and function name changes. To validate the regular expressions, we manually checked all occurrences of the function name that did not match the regular expression for a function call and found that the regular expression behaved as intended.

2. Generate stack traces

Because the stack trace is used as part of the input for the APR model, we must generate it by running the tests. A transformed function can produce a different stack trace, so we must generate the stack traces for the original and transformed functions separately. The process of producing a stack trace for a given bug is as follows:

a) Check out the buggy version of the project.

b) (If transformed) Substitute the transformed function for the original function.

c) Compile the project.

d) Run the tests.

```
Exception in thread "main" java.lang.NullPointerException
  at com.example.Main.main(Main.java:10)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(
     NativeMethodAccessorImpl.java:62)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(
     DelegatingMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke(Method.java:498)
  at com.intellij.rt.execution.application.AppMain.main(AppMain.
     java:147)
```

(a) Original stack trace

```
Exception in thread "main" java.lang.NullPointerException
  at com.example.Main.main(Main.java:10) if (a.equals(b)) return;
```

(b) Summarized stack trace

Figure 4.3: An example of stack trace summarization. The lines that do not belong to the project under test are filtered out. After this, the actual line of code is appended to the relevant stack trace line.

e) Capture the output of the failing tests.

f) Summarize this output.

To keep the prompt within context limits, the stack traces must be summarized, as they can be very long. An example of such a summarization is shown in fig. 4.3.

Note that the project will likely not compile when a function is substituted for a function with a different name. Therefore, we substitute a version of the transformed function with the original function name to run the tests. We then update the function name in the stack trace to the new name with a regular expression.

3. Generate patches

We generate the patches by querying our APR model as described in section 4.1. We use regular expressions to extract the patches from the responses.

4. Evaluate patches

We evaluate the quality of the patches with unit tests. For each of the patches, we do the following:

a) Check out the buggy version of the project.

b) Substitute the patch for the buggy function.

c) Compile the project.

d) Test the project.

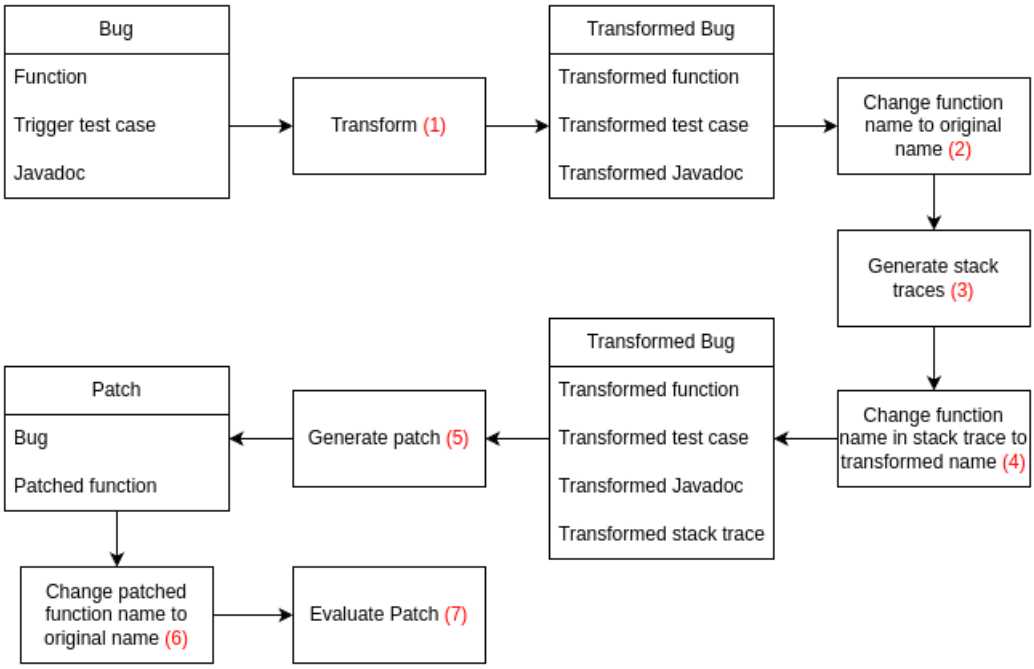

Figure 4.4: The process of transforming function names back and forth to ensure the projects compile. After transforming the buggy function, test case and Javadoc in the Transform stage (1), we change the function name in the transformed version back to the original name (2) before substituting the patch into the project, to make sure that the project can compile to generate stack traces for the trigger tests (3). However, to keep the prompt to the LLM consistent, we change the function name in the stack trace back to the transformed name (4), before prompting the LLM to generate a patch (5). Finally, we change the function name in the patched function back to the original name again (6), to evaluate whether the patch passes the tests (7).

e) Check the result (Plausible, Failing, Uncompilable, Timeout)

f) Store the result of this patch.

To make sure that the project compiles, the function name of the patched function is changed back to the original name if necessary. A full overview of the process of changing function names back and forth is shown in fig. 4.4.

Generating stack traces, generating patches, and evaluating patches for all of the bugs is a time-consuming process. To speed this up, we make heavy use of parallelism. To avoid conflicts, all operations are performed on separate project clones, and all results are written synchronously. The **Transform** and **Generate Stack traces** steps have to be executed only once to transform the dataset. The other stages have to be executed for each model under test separately.

Chapter 5

Results

5.1 RQ1: To what extent does metamorphic testing affect the performance of SOTA LLMs for code repair?

Table 5.1 shows the success rate of state-of-the-art APR models on the original and transformed versions of the Defects4J and GitBug-Java datasets. The results on Defects4J reveal that all three of these models show statistically significant performance degradation on the transformed Defects4J benchmark, supporting the hypothesis that the original results are inflated due to data leakage. ChatGPT-4o, ChatGPT-4o-mini and Claude-3.7-Sonnet perform 2.5%, 3.1% and 4.1% worse on average.

As shown in fig. 5.1, the median performance difference is -10% for each model, indicating that the transformations decrease performance on more than half of the bugs. For each model, the distribution is visibly skewed towards lower performance after transforming. The maximum performance decrease is 80%, 70% and 100% for ChatGPT-4o, ChatGPT-4o-mini, and Claude-3.7-Sonnet, while the maximum performance increase is 50%, 70% and 80% for ChatGPT-4o, ChatGPT-4o-mini, and Claude-3.7-Sonnet respec-

Table 5.1: Success rate (*SR*) of SOTA models on the Defects4J and GitBug-Java datasets and their transformed counterparts. Statistically significant bugs are marked in bold. On the Defects4J dataset, all three models perform significantly worse after applying metamorphic transformations. There are also differences in performance on the GitBug-Java dataset, but they are statistically insignificant.

Dataset	#Bugs	Model	SR_{orig}	SR_{trans}	SR_{diff}
Defects4J	401	ChatGPT-4o	46.9%	44.4%	**-2.5%**
		ChatGPT-4o-mini	29.8%	26.7%	**-3.1%**
		Claude-3.7-Sonnet	67.9%	63.8%	**-4.1%**
GitBug-Java	35	ChatGPT-4o	24.6%	26.0%	+1.4%
		ChatGPT-4o-mini	23.7%	20.3%	-3.4%
		Claude-3.7-Sonnet	63.4%	64.0%	+0.6%

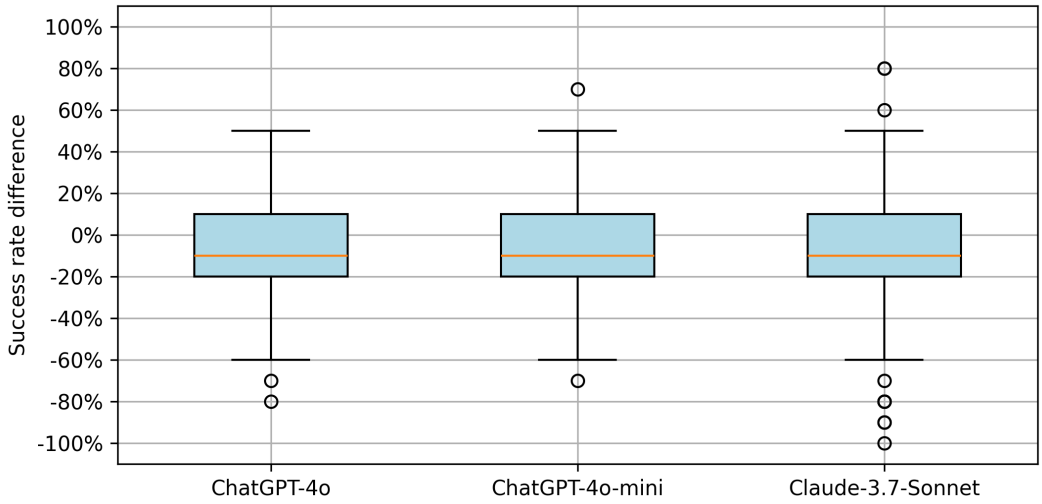

Figure 5.1: Difference in success rate (SR_{diff}) after transforming each bug of the Defects4J benchmark. For each model, the median performance is 10% lower after transforming. To highlight the differences, bugs with a 0% or 100% pass-rate for both original and transformed variants were filtered out for each model. The boxplots for ChatGPT-4o, ChatGPT-4o-mini and Claude-3.7-Sonnet include 281, 251 and 241 bugs respectively.

tively. Though these values are influenced by the non-deterministic nature of LLMs, this highlights that metamorphic transformations can have a dramatic effect on the performance of an LLM on specific bugs.

In contrast, the results are inconclusive on the GitBug-Java dataset, as also shown in table 5.1. We do not observe significant differences between the performance on the original benchmark and its transformed counterpart. This could be attributed to the smaller number of samples (n=35), which limits the statistical power of the analysis. However, this could also be because GitBug-Java contains more recent bugs and thus has a much lower risk of data leakage.

However, we can also observe that the performance differences on the GitBug-Java benchmark are not skewed towards higher or lower performance overall (as shown in fig. 5.2), with the distribution of Claude-3.7-Sonnet even being entirely symmetrical. Though some variance is shown, no overall trend can be observed.

Bugs

Table 5.2 shows the number of bugs with a significant performance difference after transforming. There is only one bug that shows a statistically significant performance difference on more than one model: Both ChatGPT-4o and Claude-3.7-Sonnet perform significantly worse on `Closure-78`. Clearly, the models have a lower success rate after transformation in most cases, as there are 18 bugs with a significantly lower success rate and only 5 bugs with a significantly higher success rate.

We conduct a case study in chapter 6, investigating why one of these significant bugs

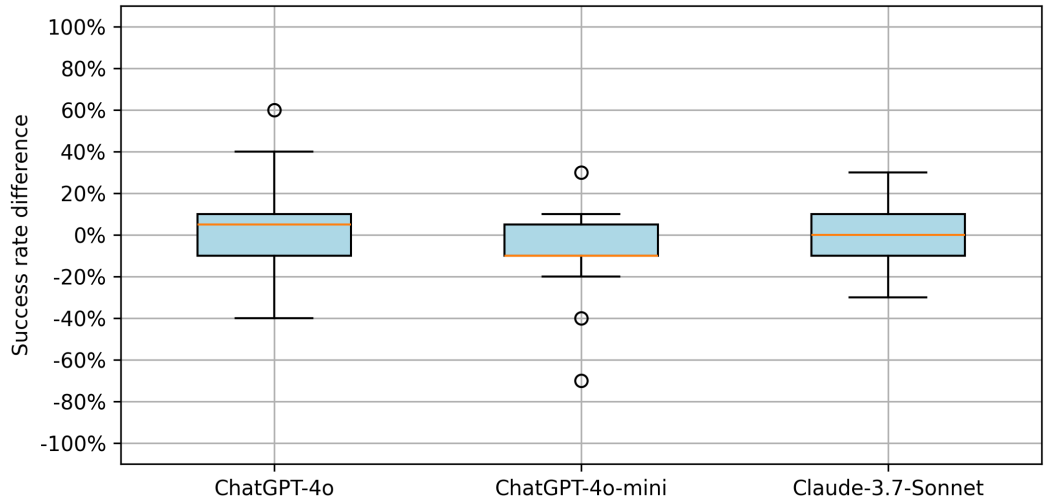

Figure 5.2: Difference in success rate (SR_{diff}) after transforming each bug of the GitBug-Java benchmark. To highlight the differences, bugs with a 0% or 100% pass-rate for both original and transformed variants were filtered out for each model. The boxplots for ChatGPT-4o, ChatGPT-4o-mini and Claude-3.7-Sonnet include 18, 15 and 26 bugs respectively.

Table 5.2: Number of bugs with a significant performance difference under metamorphic testing across SOTA models. In the majority of these cases, the performance is significantly worse after transforming the bug.

Dataset	Model	Significantly better	Significantly worse
Defects4J	ChatGPT-4o	0	5
	ChatGPT-4o-mini	1	3
	Claude-3.7-Sonnet	3	10
GitBug-Java	ChatGPT-4o	1	0
	ChatGPT-4o-mini	0	1
	Claude-3.7-Sonnet	0	0

`Lang-43` is so much harder to solve for Claude-3.7-Sonnet after transformations. This provides deeper insights and understanding of the LLM behavior and provides more explanation of our results.

Interestingly, the model performs significantly *better* on several bugs after transformations as well. It could be that the transformations enhance the readability of the code in some cases, but the number of bugs where the performance significantly increased is too low to draw conclusions.

Answer to RQ1: State-of-the-art LLMs perform significantly worse on the metamorphically transformed Defects4J benchmark. They do not suffer as much on the GitBug-Java Benchmark. Several bugs are particularly vulnerable to metamorphic transformations.

5.2 RQ2: To what extent do specific metamorphic transformations affect the performance of SOTA LLMs for code repair?

Table 5.3 shows the results of the two-way ANOVA test for testing interactions between factors. Different models appear to be vulnerable to different subsets of transformations. We observe that Claude-3.7-Sonnet is affected by a wider variety of combinations of transformations, with 15 different combinations having a significant effect on the performance of this model, compared to 9 and 5 combinations for ChatGPT-4o and ChatGPT-4o-mini respectively. Moreover, we observe that Claude-3.7-Sonnet is particularly vulnerable to identifier renaming transformations. Applying two or three kinds of renaming results in a statistically very significant($p < 0.01$) correlation for this model. Finally, 20 out of 29 of the effective combinations of transformations involve both renaming and structural/expression-level combinations. This highlights the importance of using various categories of transformations and their complementary nature.

Answer to RQ2: Different models are vulnerable to different combinations of transformations. Claude-3.7-Sonnet is vulnerable to a wide variety of transformations, in particular to identifier renaming transformations. The majority of effective combinations of transformations across models involve both structural and renaming transformations.

5.3 RQ3: To what extent can an APR performance decrease under metamorphic testing be attributed to data leakage?

Figure 5.3 shows the mean cumulative performance drop over the NLL for the four open-source models. We observe that the mean performance of Gemma 2 27B keeps decreasing as the NLL increases, in particular for the lower percentiles. This indicates a connection between the drop in APR performance and the model's NLL on the file containing the buggy code. Indeed, if we test for a correlation between the drop in performance and the NLL for all datapoints which have a sub-median NLL, we find a Spearman correlation coefficient of 0.307 (p=0.001): For lower NLL, the model is more familiar with the code and more likely to have memorized the solution, and is therefore also more severely affected under metamorphic testing. This implies that the performance decrease under metamorphic testing we observe for this model can, to some extent, be attributed to data leakage.

Table 5.3: Results of the two-way ANOVA test between SR_{diff} and various combinations of transformations. Only statistically significant results($p<0.05$) are shown. Very statistically significant results ($p<0.01$) are marked with bold.

Model	Transformation 1	Transformation 2	Transformation 3	F-statistic
ChatGPT-4o	ForToWhile	RenameVariables		0.169
	RenameFunction	SwapRelOperands		0.226
	ExpandIncrement	ForToWhile	NestElseIf	0.198
	ForToWhile	NestElseIf	RenameFunction	0.182
	ForToWhile	NestElseIf	RenameParameters	0.225
	ForToWhile	RenameParameters	ReverseIf	**0.299**
	NestElseIf	RenameVariables	SwapEqualsOperands	0.196
	RenameVariables	ReverseIf	SwapEqualsOperands	0.200
	ReverseIf	SwapEqualsOperands	SwapRelOperands	0.256
GPT-4o-mini	NestElseIf	RenameFunction		0.292
	ExpandIncrement	NestElseIf	RenameParameters	0.299
	ExpandIncrement	RenameFunction	RenameParameters	0.208
	ForToWhile	ReverseIf	SwapEqualsOperands	0.254
	NestElseIf	RenameParameters	SwapRelOperands	0.225
Claude-3.7-Sonnet	ForToWhile	ReverseIf		0.300
	ExpandIncrement	ForToWhile	ReverseIf	0.273
	ExpandIncrement	ForToWhile	SwapEqualsOperands	**0.520**
	ForToWhile	NestElseIf	RenameParameters	0.271
	ForToWhile	RenameFunction	RenameParameters	0.333
	ForToWhile	RenameFunction	RenameVariables	0.259
	ForToWhile	RenameFunction	SwapRelOperands	0.281
	ForToWhile	ReverseIf	SwapRelOperands	0.219
	NestElseIf	RenameParameters	ReverseIf	0.303
	NestElseIf	ReverseIf	SwapEqualsOperands	0.237
	RenameFunction	RenameParameters	RenameVariables	**0.452**
	RenameFunction	RenameParameters	SwapRelOperands	**0.415**
	RenameFunction	RenameVariables	SwapEqualsOperands	**0.470**
	RenameFunction	RenameVariables	SwapRelOperands	**0.462**
	RenameVariables	SwapEqualsOperands	SwapRelOperands	0.218

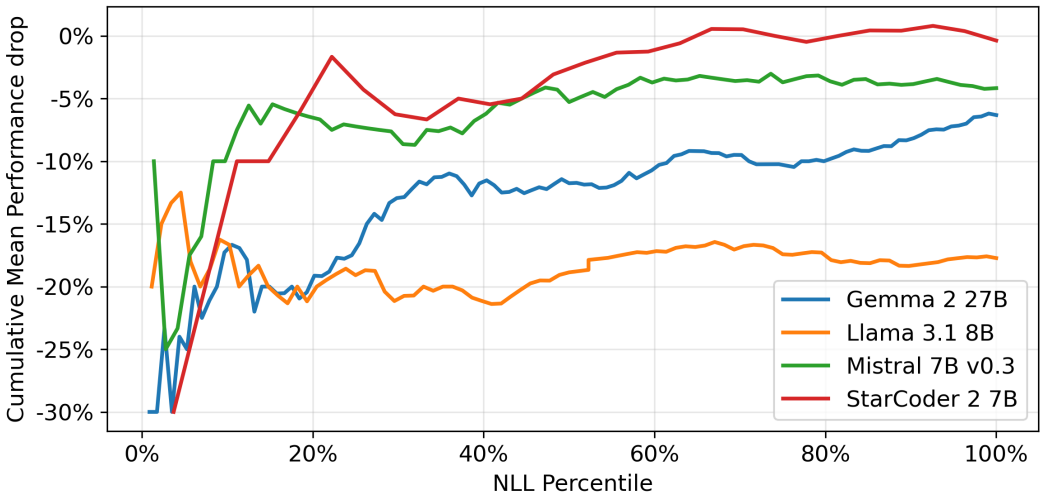

Figure 5.3: The cumulative mean performance drop over NLL percentile. To increase the interpretability of the plot, we used NLL Percentile to project our results onto a uniform distribution instead of a logarithmic distribution. Furthermore, because there is a significant amount of noise in our results due to the randomized nature of our results, we opted to display the cumulative mean performance difference over all datapoints up to the specified NLL percentile. As the plot shows, the Gemma 2 27B model shows a gradual decrease in cumulative mean as the NLL increases, particularly in the lower percentiles. In comparison, the LLama and Mistral models exhibit a relatively stable mean performance drop as the NLL increases. Finally, the Starcoder model shows a steep drop in the lowest percentiles, but quickly stabilizes after. Only bugs that could be solved by each model are shown in this plot: 228, 177, 144 and 54 bugs for the Gemma, Llama, Mistral, and Starcoder models respectively.

Many of these bugs that have a low NLL with the Gemma model are from the `Lang` and `Math` projects. As shown in fig. 5.4, these projects have both a low median NLL and a severe median success drop. In addition, the bugs from the `Codec` project also exhibit this behaviour. Both of these features suggest that the Gemma model is very familiar with these projects and that the APR performance of this model on these projects is not representative of the performance in a real-world setting.

In addition, fig. 5.4 also shows that the projects from the older Defects4J version (Version 1.2) tend to exhibit both a lower NLL and a more severe performance drop, suggesting that projects added to the Defects4J benchmark earlier tend to suffer more from data leakage.

In contrast to the datapoints in the lower percentiles, no such correlation can be observed for datapoints which have an above-median NLL, indicating that a performance drop under metamorphic testing is connected to the NLL only above a certain threshold of familiarity, i.e. if the NLL is very high, the model most likely does not remember the solution anyway and the effect of metamorphic testing is limited.

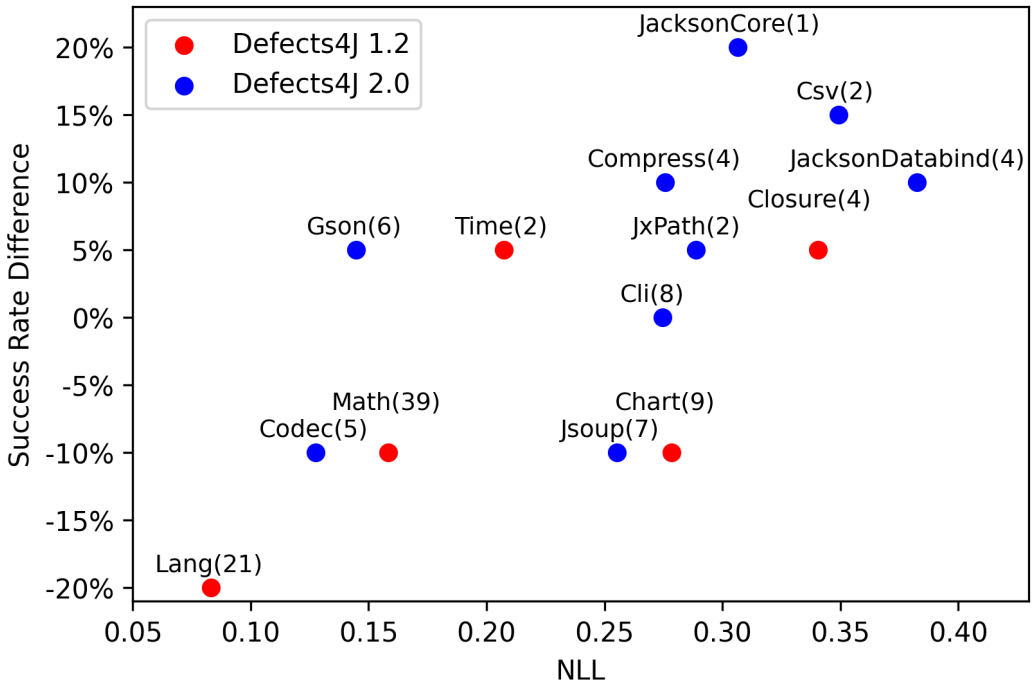

Figure 5.4: Defects4J projects by median success rate difference, median NLL, and Defects4J version. Only bugs that the Gemma 2 27B model can solve and have a sub-median NLL are included. The number of bugs left in each project is displayed next to its name in parentheses. The `Lang`, `Codec`, and `Math` projects exhibit both low median NLL and a severe median success rate drop. In general, projects from Defects4J 1.2 tend to have a lower NLL and more severe performance drop than those in Defects4J 2.0.

There is no significant correlation between the performance drop we observe due to metamorphic testing and the NLL for the other three models. As shown in fig. 5.3, the cumulative mean drop appears relatively stable over the increase in NLL for the Llama 3.1 8B and Mistral 7B v0.3 models. This does not point to any connection between data leakage and the observed performance drop under metamorphic testing for these models. For Starcoder 2 7B, we do observe a similar trend to Gemma 2 27B. However, due to the limited number of bugs the model could solve (54) and the noisy nature of the data, no significant correlation can be observed.

Answer to RQ3: Our results show a significant correlation between the negative log-likelihood and the performance decrease under metamorphic testing for the Gemma 2 27B model. This indicates that the performance decrease can, to some extent, be attributed to data leakage. We find that, in particular, the `Lang`, `Math`, and `Codec` projects exhibit clear signs of memorization behaviour.

Chapter 6

Case study: Lang-43 and Evidence of Memorization

We observe behavior consistent with our hypothesis that metamorphic testing can reveal and mitigate data leakage effects, for the bug `Lang-43` specifically. Claude-3.7-Sonnet can pass the tests with the original version of the bug 7 out of 10 times, but cannot solve the transformed version of the bug at all. While interesting LLM behavior may be observable on multiple bugs, we conduct a thorough case study on the LLM behavior for this bug to highlight the issues and provide deeper insights into the working of LLMs under metamorphic testing. To better understand how this behavior is demonstrated in practice, we conduct an in-depth study, investigating the nature of this bug, how transformations impact the syntax of the bug, and how the Claude-3.7-Sonnet behaves when solving this bug.

6.1 Bug description

`Lang-43` is a bug in the `commons-lang` project by Apache [103]. The purpose of this project is to provide utilities for many of the standard `java.lang` objects, such as `String`, `Object` `Number`, etc. The project was first released in 2002, and has become a well-known dependency for many projects. The source code is included in popular LLM pre-training benchmarks, such as TheStack [29], and is referenced very frequently on the web.

The bug `Lang-43` is located in the `appendQuotedString` method in the `Extended-MessageFormat` class. This class contains functionality for formatting strings and dynamically inserting variables into pre-defined template strings [104]. The responsibility of the `appendQuotedString` method is to consume a quoted string and append it to a given `StringBuffer`. The implementation of this method is shown in fig. 6.1. In addition, when there are two consecutive single quotes (`''`), this indicates that a single quote is escaped by another single quote, and that one of the two quotes should be appended to the `StringBuffer`. The `if` statement in lines 8-10 is responsible for this.

The problem with this code is that the `if`-statement on lines 8-10 appends a quote and returns without increasing the `ParsePosition`. This means that the current index of the character being parsed is not updated. Consequently, the method `applyPattern`, which is

```
1  private StringBuffer appendQuotedString(
2                          String pattern,
3                          ParsePosition pos,
4                          StringBuffer appendTo,
5                          boolean escapingOn) {
6    int start = pos.getIndex();
7    char[] c = pattern.toCharArray();
8    if (escapingOn && c[start] == QUOTE) {
9      return appendTo == null ? null : appendTo.append(QUOTE);
10   }
11   int lastHold = start;
12   for (int i = pos.getIndex(); i < pattern.length(); i++) {
13     if (escapingOn && pattern.substring(i).startsWith(
         ESCAPED_QUOTE)) {
14       appendTo.append(c, lastHold, pos.getIndex() - lastHold).
           append(QUOTE);
15       pos.setIndex(i + ESCAPED_QUOTE.length());
16       lastHold = pos.getIndex();
17       continue;
18     }
19     switch (c[pos.getIndex()]) {
20       case QUOTE:
21         next(pos);
22         return appendTo == null ? null : appendTo.append(c,
             lastHold, pos.getIndex() - lastHold);
23       default:
24         next(pos);
25     }
26   }
27   throw new IllegalArgumentException("Unterminated quoted string
       at position " + start);
28 }
```

Figure 6.1: The function containing the Lang-43 bug.

calling `appendQuotedString` upon encountering a quote, will encounter the same quote again and call `appendQuotedString` again. Because of this, an infinite amount of quotes is appended to the `StringBuffer` and eventually leads to an `OutOfMemoryError`.

The human-written patch, as applied in the project, involves increasing the `ParsePosition` by 1, by adding `next(pos);` at line 9, as shown in fig. 6.2. This is the only edit required to fix the bug.

6.1.1 Transformed Bug

We applied metamorphic transformations to this method and created the version shown in fig. 6.3. We applied function renaming, parameter renaming, and local variable renaming. This alone leads to a significantly different syntax, even though the semantic meaning of

```
1  private StringBuffer appendQuotedString(
2                          String pattern,
3                          ParsePosition pos,
4                          StringBuffer appendTo,
5                          boolean escapingOn) {
6    int start = pos.getIndex();
7    char[] c = pattern.toCharArray();
8    if (escapingOn && c[start] == QUOTE) {
9      next(pos);
10     return appendTo == null ? null : appendTo.append(QUOTE);
11   }
12   /* Rest of method remains identical */
13 }
```

Figure 6.2: The fix for the Lang-43 bug.

all the identifiers remains the same. Furthermore, the for loop is transformed to a while loop, the unary increments are expanded, and relational and equals operands are swapped in several positions. The transformed method's behavior and semantics remain identical to those of the original method.

6.2 LLM behavior

When prompting Claude-3.7-Sonnet, it misjudges the root cause for all 10 samples with the original method, and also in all 10 samples with the transformed method. It explains that the root cause is that the variable `i` is not updated with the `ParsePosition`. However, this is false, as the issue lies elsewhere: It lies in the `if`-statement in lines 8-10. In only one of the samples with the original non-transformed method, Claude-3.7-Sonnet mentions the particular issue of that `if`-statement in addition to the (wrong) root cause. It does not identify this issue at all in the transformed code.

In each sample with the original method, Claude-3.7-Sonnet attempts to fix the bug by updating the `i` variable with the `ParsePosition` in various ways. Interestingly, it also updates the problematic `if`-statement in 6 out of 10 cases, **without any explanation**. It adds either `next(pos);` or `pos.setIndex(startIdx + 1);` (which is identical to the implementation of `next()`) at line 9, which solves the bug. This clearly indicates memorization.

We do not observe this behavior when we prompt Claude-3.7-Sonnet with the transformed variant. It only updates the problematic `if`-statement in one sample, but also moves the `return` statement to a newly created `else` branch. Other than this, it does not touch lines 8-10 and only modifies the rest of the method.

6.3 Implications

These observations suggest the presence of data leakage; Claude-3.7-Sonnet has memorized the solution to this bug to some extent. If we examine the reasoning for identifying the root

```
1  private StringBuffer appendEscapedString(
2                          String template,
3                          ParsePosition position,
4                          StringBuffer resultBuffer,
5                          boolean isEscapingEnabled) {
6    int beginning = position.getIndex();
7    char[] c = template.toCharArray();
8    if (isEscapingEnabled && QUOTE == c[beginning]) {
9      return null == resultBuffer ? null : resultBuffer.append(QUOTE
         );
10   }
11   int lastIndex = beginning;
12   int index = position.getIndex();
13   while (template.length() > index) {
14     if (isEscapingEnabled && template.substring(index).startsWith(
         ESCAPED_QUOTE)) {
15       resultBuffer.append(c, lastIndex, position.getIndex() -
           lastIndex).append(QUOTE);
16       position.setIndex(index + ESCAPED_QUOTE.length());
17       lastIndex = position.getIndex();
18       index += 1;
19       continue;
20     }
21     switch(c[position.getIndex()]) {
22       case QUOTE:
23         next(position);
24         return null == resultBuffer ? null : resultBuffer.append(c
             , lastIndex, position.getIndex() - lastIndex);
25       default:
26         next(position);
27     }
28       index += 1;
29     }
30   throw new IllegalArgumentException("Unterminated quoted string
       at position " + beginning);
31 }
```

Figure 6.3: The transformed version of the Lang-43 bug.

cause of the bug, we observe that Claude-3.7-Sonnet can, in general, not find the root cause. Therefore, we should conclude that Claude-3.7-Sonnet cannot solve this bug. However, when writing the patches, adding a line of code that increases the `ParsePosition` at line 9 is a frequent occurrence, even though this line is not in the buggy code snippet, and the model does not explicitly reason toward this.

When asking Claude-3.7-Sonnet to solve the transformed variant of the bug, adding code to increase the `ParsePosition` is not as frequent. We hypothesize that the metamorphic transformations disturb the context representation in a way that the model is less inclined to add this code.

Although each of these patches would be rejected by a human semantics review, this example illustrates the data leakage problem and how this can inflate the results. Claude-3.7-Sonnet cannot correctly identify the root cause and suggest solutions for this bug, and can most likely only suggest patches that pass the tests because of memorization. Such cases should not be classified as correct, and doing so will lead to invalid results.

This is a deep dive into one specific case, but as shown in chapter 5, there is a general trend that the models perform worse after transformations. Most likely, a similar effect can be observed in the other bugs.

Chapter 7

Discussion

This chapter discusses the implications of our findings, placing them in a context with prior works, and considering the robustness and limitations of our methodology. We reflect on the effectiveness of metamorphic testing in revealing and potentially mitigating data leakage in LLMs for APR, outline threats to the validity of our conclusions, and provide recommendations and directions for future work.

7.1 Interpretation of findings

Our results reveal that even state-of-the-art LLMs for code repair exhibit a lack of robustness to natural, semantic-preserving, metamorphic transformations. We observe that all three APR models under test (ChatGPT-4o, ChatGPT-4o-mini, and Claude-3.7-Sonnet) perform significantly worse on a metamorphically transformed Defects4J benchmark. These findings emphasize that even the best-performing, state-of-the-art models are vulnerable to simple semantic-preserving code changes and lack robustness.

In addition, the models do not just suffer from unnecessary prediction flips, where a model changes its output due to metamorphic transformations, but we also observe that the models tend to change from correct to wrong more frequently than vice versa. This implies that the original bug is inherently easier for the model to solve than the transformed bug. A plausible explanation for this is that the model remembers the solution to the original bug, but can remember the solution to the transformed bug less reliably, and is forced to rely more on reasoning.

The models do not show any statistically significant performance difference on the transformed GitBug-Java Benchmark. This may be due to two things. First, the sample size is much smaller, as there are only 35 bugs of appropriate complexity in the GitBug-Java dataset, compared to the 401 in Defects4J. A smaller sample size naturally leads to less statistical power. Second, the GitBug-Java dataset is a much more recent dataset, designed specifically with data leakage mitigation in mind. Due to this, the models may not be as familiar with the original projects and thus not memorize as much, leading to less significant differences.

We also find that Claude-3.7-Sonnet is particularly vulnerable to transformations that

involve identifier renaming, even though we replace identifiers with semantically similar alternatives. This is another observation that can be explained by data leakage. Identifier names can be cues for the model to remember the solution to the bug. When these are changed, the bug is much harder to memorize. The fact that we observe this mostly for Claude-3.7-Sonnet can be explained by the fact that larger models tend to memorize more [72].

Finally, we investigated whether a decrease in performance under metamorphic testing is associated with other estimations of data leakage: NLL. We found a significant correlation between these metrics for the Gemma 2 27B model, which is compelling evidence for our hypothesis that metamorphic testing can reveal and mitigate data leakage: if a model is very familiar with a piece of code, metamorphic testing leads to a greater performance reduction. This implies that the performance is inflated due to memorization and that the model struggles to remember the solution to the bugs when they are transformed.

To summarize, our results provide additional evidence for the prevalent problem of data leakage in LLMs. Many reported results of LLM-based APR tools are inflated and do not generalize to unseen problems. Although significant effort has been put into mitigating the effects of data leakage, our findings suggest that these issues persist, even in models as advanced as Claude-3.7-Sonnet and ChatGPT-4o.

7.2 Comparison with prior work

Our findings align with prior works showing the vulnerability of deep code models. As shown in [14], [15], [77], [105]–[107], deep code models are not robust to metamorphic transformations. This holds specifically for APR models as well, as shown in [78], [79], [88]. We show that this holds for state-of-the-art LLMs for APR as well. In addition, our results align with the findings in [87]: APR results do not generalize well to Defects4J variant benchmarks.

Furthermore, our findings align with the ideas presented by Ramos et al. [18], suggesting that LLMs suffer from data leakage on the Defects4J [11] dataset. While we do not provide direct evidence of data leakage, our results support the idea that the reported performance on the original dataset is inflated. Although we only found a connection between the NLL and performance drop for one model: Gemma 2 27B, this aligns with their findings that Gemma 2 27B suffers the most from data leakage out of the models we tested. In addition, Gemma 2 27B is the largest out of the models we tested for a data leakage correlation, and larger models tend to memorize more [72].

Finally, our work has explored suggestions made by Sallou et al. [17] and shown the potential of using metamorphic transformations to mitigate the effects of data leakage. It appears that metamorphic testing can, to some extent, mitigate the effects of data leakage. Our work provides new insights into a new technique for data leakage detection and mitigation. However, further research is required to understand the full potential of this technique and how effective it can be.

7.3 Threats to validity

We acknowledge that several threats could compromise the validity of our study. We categorize these threats into four groups:

- **Internal threats** are related to our implementations.
- **External threats** are related to artifacts that are not directly related to our implementations.
- **Construct threats** are related to the metrics and how reliably they capture the quality we intend to measure.

7.3.1 Threats to internal validity

First of all, the implementation may contain bugs, resulting in unnatural or even semantically different metamorphic variants. To combat this threat, we extensively tested our implementation with both unit tests and manual samples. Second, the experimental pipeline could contain implementation bugs. However, unless bugs are present in the transform stage, these bugs affect both original and transformed variants equally. Nevertheless, every stage of the pipeline was manually verified to be correct. Third, we update the function names in the test and stack traces using regular expressions. These regular expressions may not be perfect and may miss function references. We manually verified that every occurrence of the function name that was not matched by the regular expression did not actually reference the function. Examples of this would be if the function name is a substring of another variable name or a substring of the name of the test.

Finally, metamorphic transformations could result in unnatural code snippets, putting the APR models at a disadvantage, as they were mostly trained on human-written code. This would also be an explanation of the observed performance drop. However, we only chose natural transformations [77], [78] and used an LLM to generate meaningful synonyms for identifier names. Related to this, the NLL is an indication of how natural a piece of code appears to the model [18], and is therefore also strongly correlated with code naturalness to a human. This could pose a risk to the validity of our conclusions. However, the NLL value only reflects the perceived naturalness of the **original** code, and is not affected by any possible kind of naturalness difference introduced by the transformations.

7.3.2 Threats to external validity

The main threat to external validity is the results provided by Ramos et al. [18]. Their dataset only includes a subset of the Defects4J dataset, containing only the `Closure`, `Lang`, `Chart`, `Math`, and `Mockito` projects. This is not completely representative of the full dataset, so we reproduced their method on the remainder of the benchmark to collect more samples. However, their implementation may also contain bugs, or their assumptions may be invalid.

Furthermore, all results are only measured in the context of automated program repair. The concept of using metamorphic transformations to mitigate data leakage may not gener-

alize to other settings. Despite this, we have no reason to believe that this concept does not apply to other coding tasks, such as code or unit test generation.

7.3.3 Threats to construct validity

Our main threat to construct validity is our metric for APR model performance. We judge the performance of an APR model based on how frequently it can generate a patch that passes the test suites. However, *test overfitting* is a well-known problem in the field of APR. This occurs when a model generates a patch that passes the test suite, but does not fix the original bug completely, overfitting on the test suite [75]. Many papers use exact-match metrics [3], [42], [108]–[110] or manual correctness checking [5], [6], [8], [64], [111]. We opted to only use the test suite pass rate as our metric, as exact-match metrics tend to suffer when bugs have multiple repair solutions [3] and manual correctness checking requires an infeasible amount of manual effort. Even though the test suite metric is not optimal, it is identical across both the original and transformed groups in our experiment. It is unlikely that patch overfitting is more prevalent in the original group than in the transformed group.

Another threat to construct validity is the way we measure the effect of different transformations. Transformations are applied at every opportunity where they can be applied. Therefore, we do not only compute the correlation between the performance drop and the transformations, but also the correlation between the performance drop and the size/complexity of a bug. For example, if a variable name transformation was applied once to bug A and three times to bug B, this also means that bug B has three times as many local variables. This means that our tests may appear to indicate the effect of transformations, but show the effectiveness of transformations in general on code snippets with specific 'transformable' features.

7.4 Recommendations

We recommend that researchers report results on metamorphically transformed benchmarks when evaluating the performance of LLM-based tools. This has two distinct functions. Firstly, this gives insight into the robustness of an LLM-based tool. It has been shown that many LLM-based tools lack robustness [14], [15], [77], [105]–[107]. Showing the performance on a transformed benchmark ensures that readers and users can more confidently apply it to other scenarios.

Secondly, significant differences across the original and transformed benchmarks may indicate data leakage issues. Especially in cases where it is not 100% certain that the training set and benchmark do not overlap, showing that the results are not inflated due to data leakage is essential to prove the validity of the results.

7.5 Future work

With the rise of LLM agents that can operate autonomously in a complete software project [7], [112]–[115], it is critical that these can be evaluated with metamorphic testing as well.

At this point, CodeCocoon applies only to isolated Java functions and cannot use any of the class context. In our case, we were able to test with a function renaming transformation because we were able to transform the function names back before running the tests. However, such techniques are much more difficult to apply when testing an agentic system. A future iteration of CodeCocoon should apply metamorphic transformations to projects as a whole.

Furthermore, Python is currently the most prevalent programming language [116]. CodeCocoon only applies to Java code snippets, but should be made to transform Python code as well. This could require a significant amount of effort, as one could argue that Python is a much less structured language than Java, and that this may introduce extra caveats and issues. For example, dynamic typing may introduce ambiguities in the type of objects being used, and may thus introduce issues when renaming a member of such an object. In some cases, it may be undecidable whether to change code features. This requires careful and intensive consideration when designing a Python version of CodeCocoon.

Moreover, to do a more reliable robustness evaluation, CodeCocoon should be extended to apply transformations in a non-deterministic manner. With this, we can create a set of variants instead of only one transformed version of the code. This set of variants can more accurately represent the various coding styles employed by different software engineers and give more reliable insight into the robustness of tools.

In addition, it appears that metamorphic transformations can significantly *increase* the APR success rate for some of the bugs. It could be that the transformations enhance the readability of the code in some cases, but the number of cases where the performance significantly increased is too small to derive strong conclusions. Future works with more extensive datasets should investigate these cases in more detail.

Even though our findings suggest that CodeCocoon can mitigate data leakage to some extent, further research must be conducted into what code features lead to LLMs recognizing pieces of code and what transformations mask these features best. The transformations in this study were chosen based on other criteria, and a future system with transformations specifically designed for masking recognition-critical features could lead to more effective data leakage mitigation.

As a concrete experiment, future research could systematically evaluate data leakage under metamorphic testing by training two models, where the evaluation benchmark is included in the pre-training dataset for one model, but not the other. Then, metamorphic transformations can be tested specifically for data leakage mitigation potential. The desired effect of these metamorphic transformations is that they only affect the model with data leakage and do not affect the model without data leakage. This allows for a systematic evaluation of different metamorphic transformations and their data leakage mitigation potential.

Finally, more research is required to understand data leakage and how it impacts evaluation results. As explained in chapter 6, LLMs do not always reproduce the original patch verbatim, but can remember the patch on a deeper level, sometimes reproducing a patch with a different, yet equivalent piece of code. In addition to this, more research is required to understand how metamorphic testing can affect memorization. Although the syntax of the code is very different after transforming, the embedded representations may be more

similar, which could lead to less effective data leakage mitigation. This should be explored in future research to determine the true potential of applying metamorphic testing for data leakage mitigation.

Chapter 8

Conclusion

Effective automated program repair methods can lead to massive cost reductions and have improved a lot in recent times. However, reliable and representative evaluations that reflect the true performance of systems in real-world scenarios are critical for progress in this field. The validity of evaluations as they are conducted at this point in time is at risk due to the phenomenon of data leakage. This occurs when LLMs remember the solutions to benchmark problems rather than solving them by reasoning alone, leading to inflated results and a false sense of progress.

In this study, we examine the potential of using metamorphic transformations to mitigate the effects of data leakage. For this, we create a variant benchmark for popular, well-established benchmarks Defects4J and GitBug-Java, and evaluate the APR performance of several LLMs on these benchmarks and their transformed counterparts. In addition, we investigate the effect of different combinations of metamorphic transformations on the performance of these models. Finally, we investigate to what extent our results align with data leakage metrics from other studies.

Our results show that state-of-the-art LLMs for code repair exhibit significant performance degradation on a metamorphically transformed Defecsts4J benchmark. The performance drops by 2.5%, 3.1%, 4.1% for ChatGPT-4o, ChatGPT-4o-mini and Claude-3.7-Sonnet respectively. This highlights the lack of robustness in these models and suggests that the results on the original benchmark may be inflated due to data leakage.

Moreover, we have shown that renaming transformations significantly contribute to the performance drop for Claude-3.7-Sonnet. The observation that renaming identifiers with meaningful alternatives is so detrimental to the performance of one of the best-performing LLMs for code repair also suggests that these variable names are cues for the model to memorize the solution to a bug.

Furthermore, we find that there is a significant connection between the observed performance drop and *negative log-likelihood* for the Gemma 2 27B model. This indicates that metamorphic testing leads to a particularly severe performance drop on bugs that have a high risk of memorization. The observed connection further demonstrates the potential of metamorphic testing for data leakage detection and mitigation.

In addition to this, we provide further insights as to how metamorphic testing affects LLM behavior in a case study. We show that the LLM can, in this case, apply the correct

patch by memorization rather than reasoning. It is not able to apply the correct patch as reliably on the transformed variant, supporting our hypothesis that metamorphic testing can reveal and mitigate data leakage issues.

In summary, many results of LLM-based tools on the original benchmarks are inflated and do not generalize to unseen data. This is a major threat to the validity of evaluations and could become more severe as LLMs are trained on increasingly larger training sets. Mitigation techniques for data leakage require urgent attention.

Collecting a new benchmark from more recent projects appears to reduce data leakage issues to some extent, as shown by the fact that there is no significant performance difference between the original and transformed versions of the GitBug-Java dataset. However, the collection process requires intensive manual effort and is not sustainable with the rapid rate of LLM updates.

Thus, we recommend that researchers report performance on both original and transformed benchmarks when evaluating LLM-based tools. Our findings suggest that transforming an existing benchmark could be a cheap and fast alternative for collecting a new dataset. Reporting performance on both benchmarks gives insights into the robustness of the tool and can also provide confidence that results are not overly inflated due to data leakage.

Finally, further research should investigate whether our findings generalize to other models, tasks, and benchmarks. Although we have no reason to believe our results do not generalize, there still should be a more thorough evaluation in different contexts. In addition, the extent to which data leakage can be mitigated by metamorphic transformations should be explored. With transformations more targeted towards code features that elicit memorization, data leakage mitigation via metamorphic transformations could be more effective.

Bibliography

[1] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, New York, NY, USA: Association for Computing Machinery, May 27, 2018, pp. 572–583, ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180175. [Online]. Available: https://dl.acm.org/doi/10.1145/3180155.3180175 (visited on 03/13/2025).

[2] H. Krasner, "The cost of poor software quality in the us: A 2020 report," Consortium for Information & Software Quality (CISQ), USA, Tech. Rep., Jan. 2021, Member, Advisory Board. [Online]. Available: https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf.

[3] K. Huang *et al.*, *A survey on automated program repair techniques*, May 13, 2023. DOI: 10.48550/arXiv.2303.18184. arXiv: 2303.18184[cs]. [Online]. Available: http://arxiv.org/abs/2303.18184 (visited on 02/15/2025).

[4] Q. Zhang *et al.*, *A systematic literature review on large language models for automated program repair*, May 12, 2024. DOI: 10.48550/arXiv.2405.01466. arXiv: 2405.01466[cs]. [Online]. Available: http://arxiv.org/abs/2405.01466 (visited on 02/15/2025).

[5] J. Xiang *et al.*, *How far can we go with practical function-level program repair?* Oct. 31, 2024. DOI: 10.48550/arXiv.2404.12833. arXiv: 2404.12833. [Online]. Available: http://arxiv.org/abs/2404.12833 (visited on 11/14/2024).

[6] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024, New York, NY, USA: Association for Computing Machinery, Sep. 11, 2024, pp. 819–831, ISBN: 9798400706127. DOI: 10.1145/3650212.3680323. [Online]. Available: https://dl.acm.org/doi/10.1145/3650212.3680323 (visited on 11/15/2024).

[7] I. Bouzenia, P. Devanbu, and M. Pradel, *RepairAgent: An autonomous, LLM-based agent for program repair*, Oct. 28, 2024. DOI: 10.48550/arXiv.2403.17134. arXiv: 2403.17134[cs]. [Online]. Available: http://arxiv.org/abs/2403.17134 (visited on 04/22/2025).

[8] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, New York, NY, USA: Association for Computing Machinery, Nov. 30, 2023, pp. 172–184, ISBN: 9798400703270. DOI: 10.1145/3611643.3616271. [Online]. Available: https://dl.acm.org/doi/10.1145/3611643.3616271 (visited on 04/22/2025).

[9] F. Li, J. Jiang, J. Sun, and H. Zhang, *Hybrid automated program repair by combining large language models and program analysis*, Jun. 4, 2024. DOI: 10.48550/arXiv.2406.00992. arXiv: 2406.00992[cs]. [Online]. Available: http://arxiv.org/abs/2406.00992 (visited on 03/15/2025).

[10] L. Vidziunas, D. Binkley, and L. Moonen, "The impact of program reduction on automated program repair," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ISSN: 2576-3148, Oct. 2024, pp. 337–349. DOI: 10.1109/ICSME58944.2024.00039. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10795018 (visited on 03/15/2025).

[11] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, New York, NY, USA: Association for Computing Machinery, Jul. 21, 2014, pp. 437–440, ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2628055. [Online]. Available: https://dl.acm.org/doi/10.1145/2610384.2628055 (visited on 02/14/2025).

[12] Z. Chen, J. M. Zhang, M. Hort, M. Harman, and F. Sarro, "Fairness testing: A comprehensive survey and analysis of trends," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, 137:1–137:59, Jun. 4, 2024, ISSN: 1049-331X. DOI: 10.1145/3652155. [Online]. Available: https://dl.acm.org/doi/10.1145/3652155 (visited on 03/18/2025).

[13] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 1–36, Jan. 2022, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.2019.2962027. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9000651 (visited on 03/18/2025).

[14] L. Applis, A. Panichella, and A. van Deursen, "Assessing robustness of ML-based program analysis tools using metamorphic program transformations," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ISSN: 2643-1572, Nov. 2021, pp. 1377–1381. DOI: 10.1109/ASE51524.2021.

9678706. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9678706 (visited on 09/07/2024).

[15] L. Applis, A. Panichella, and R. Marang, "Searching for quality: Genetic algorithms and metamorphic testing for software engineering ML," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '23, New York, NY, USA: Association for Computing Machinery, Jul. 12, 2023, pp. 1490–1498, ISBN: 9798400701191. DOI: 10.1145/3583131.3590379. [Online]. Available: https://dl.acm.org/doi/10.1145/3583131.3590379 (visited on 09/10/2024).

[16] K. Zhou *et al.*, *Don't make your LLM an evaluation benchmark cheater*, Nov. 3, 2023. DOI: 10.48550/arXiv.2311.01964. arXiv: 2311.01964[cs]. [Online]. Available: http://arxiv.org/abs/2311.01964 (visited on 02/15/2025).

[17] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: The threats of using LLMs in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER'24, New York, NY, USA: Association for Computing Machinery, May 24, 2024, pp. 102–106, ISBN: 9798400705007. DOI: 10.1145/3639476.3639764. [Online]. Available: https://dl.acm.org/doi/10.1145/3639476.3639764 (visited on 02/18/2025).

[18] D. Ramos, C. Mamede, K. Jain, P. Canelas, C. Gamboa, and C. L. Goues, *Are large language models memorizing bug benchmarks?* Nov. 30, 2024. DOI: 10.48550/arXiv.2411.13323. arXiv: 2411.13323[cs]. [Online]. Available: http://arxiv.org/abs/2411.13323 (visited on 01/22/2025).

[19] A. Silva, N. Saavedra, and M. Monperrus, "GitBug-java: A reproducible benchmark of recent java bugs," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24, New York, NY, USA: Association for Computing Machinery, Jul. 2, 2024, pp. 118–122, ISBN: 9798400705878. DOI: 10.1145/3643991.3644884. [Online]. Available: https://dl.acm.org/doi/10.1145/3643991.3644884 (visited on 04/25/2025).

[20] Y. Wu, Z. Li, J. M. Zhang, and Y. Liu, "ConDefects: A complementary dataset to address the data leakage concern for LLM-based fault localization and program repair," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024, New York, NY, USA: Association for Computing Machinery, Jul. 10, 2024, pp. 642–646, ISBN: 9798400706585. DOI: 10.1145/3663529.3663815. [Online]. Available: https://dl.acm.org/doi/10.1145/3663529.3663815 (visited on 02/15/2025).

[21] Q. Zhang *et al.*, *A critical review of large language model on software engineering: An example from ChatGPT and automated program repair*, Apr. 17, 2024. DOI: 10.48550/arXiv.2310.08879. arXiv: 2310.08879[cs]. [Online]. Available: http://arxiv.org/abs/2310.08879 (visited on 02/15/2025).

[22] X. Zhou *et al.*, *LessLeak-bench: A first investigation of data leakage in LLMs across 83 software engineering benchmarks*, Feb. 10, 2025. DOI: 10.48550/arXiv.2502.06215. arXiv: 2502.06215[cs]. [Online]. Available: http://arxiv.org/abs/2502.06215 (visited on 02/16/2025).

[23] OpenAI, *Chatgpt-4o*, 2024. [Online]. Available: https://openai.com.

[24] OpenAI, *Chatgpt-4o-mini*, 2024. [Online]. Available: https://openai.com.

[25] Anthropic, *Claude 3.7 sonnet release note*, https://www.anthropic.com/news/claude-3-7-sonnet-release, Accessed: 2025-04-23, 2025.

[26] A. Grattafiori *et al.*, *The llama 3 herd of models*, Nov. 23, 2024. DOI: 10.48550/arXiv.2407.21783. arXiv: 2407.21783[cs]. [Online]. Available: http://arxiv.org/abs/2407.21783 (visited on 03/16/2025).

[27] G. Team *et al.*, *Gemma 2: Improving open language models at a practical size*, Oct. 2, 2024. DOI: 10.48550/arXiv.2408.00118. arXiv: 2408.00118[cs]. [Online]. Available: http://arxiv.org/abs/2408.00118 (visited on 03/16/2025).

[28] A. Q. Jiang *et al.*, *Mistral 7b*, Oct. 10, 2023. DOI: 10.48550/arXiv.2310.06825. arXiv: 2310.06825[cs]. [Online]. Available: http://arxiv.org/abs/2310.06825 (visited on 03/16/2025).

[29] A. Lozhkov *et al.*, *StarCoder 2 and the stack v2: The next generation*, Feb. 29, 2024. DOI: 10.48550/arXiv.2402.19173. arXiv: 2402.19173[cs]. [Online]. Available: http://arxiv.org/abs/2402.19173 (visited on 03/16/2025).

[30] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, New York, NY, USA: Association for Computing Machinery, Aug. 12, 2019, pp. 302–313, ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338911. [Online]. Available: https://doi.org/10.1145/3338906.3338911 (visited on 02/17/2025).

[31] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug. 2016, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.2016.2521368. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7390282 (visited on 02/18/2025).

[32] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan. 2012, ISSN: 1939-3520. DOI: 10.1109/TSE.2011.104. [Online]. Available: https://ieeexplore.ieee.org/document/6035728 (visited on 04/22/2025).

[33] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, New York, NY, USA: Association for Computing Machinery, Jun. 3, 2015, pp. 43–54, ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737988. [Online]. Available: https://dl.acm.org/doi/10.1145/2737924.2737988 (visited on 04/23/2025).

[34] Y. Yuan and W. Banzhaf, "ARJA: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, Oct. 2020, ISSN: 1939-3520. DOI: 10.1109/TSE.2018.2874648. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8485732 (visited on 04/23/2025).

[35] Y. Wei *et al.*, "Automated fixing of programs with contracts," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10, New York, NY, USA: Association for Computing Machinery, Jul. 12, 2010, pp. 61–72, ISBN: 978-1-60558-823-0. DOI: 10.1145/1831708.1831716. [Online]. Available: https://dl.acm.org/doi/10.1145/1831708.1831716 (visited on 04/22/2025).

[36] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts: An extended study," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2841–2857, Dec. 2021, ISSN: 1939-3520. DOI: 10.1109/TSE.2020.2970009. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8972483 (visited on 04/23/2025).

[37] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, May 18, 2013, pp. 772–781, ISBN: 978-1-4673-3076-3. (visited on 04/23/2025).

[38] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, May 2013, pp. 802–811. DOI: 10.1109/ICSE.2013.6606626. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6606626 (visited on 04/22/2025).

[39] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, Mar. 2016, pp. 213–224. DOI: 10.1109/SANER.2016.76. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7476644 (visited on 04/23/2025).

[40] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, Florence, Italy: IEEE Press, May 16, 2015, pp. 471–482, ISBN: 978-1-4799-1934-5. (visited on 04/23/2025).

[41] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, 19:1–19:29, Sep. 2, 2019, ISSN: 1049-331X. DOI: 10.1145/3340544. [Online]. Available: https://dl.acm.org/doi/10.1145/3340544 (visited on 03/05/2025).

[42] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, Sep. 2021, ISSN: 1939-3520. DOI: 10.1109/TSE.2019.2940179. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8827954 (visited on 04/23/2025).

[43] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, *Neural program repair by jointly learning to localize and repair*, Apr. 3, 2019. DOI: 10.48550/arXiv.1904.01720. arXiv: 1904.01720[cs]. [Online]. Available: http://arxiv.org/abs/1904.01720 (visited on 04/23/2025).

[44] S. Minaee *et al.*, *Large language models: A survey*, Feb. 20, 2024. DOI: 10.48550/arXiv.2402.06196. arXiv: 2402.06196[cs]. [Online]. Available: http://arxiv.org/abs/2402.06196 (visited on 03/17/2025).

[45] H. Naveed *et al.*, *A comprehensive overview of large language models*, Oct. 17, 2024. DOI: 10.48550/arXiv.2307.06435. arXiv: 2307.06435[cs]. [Online]. Available: http://arxiv.org/abs/2307.06435 (visited on 03/17/2025).

[46] A. Vaswani *et al.*, *Attention is all you need*, Aug. 2, 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762[cs]. [Online]. Available: http://arxiv.org/abs/1706.03762 (visited on 03/17/2025).

[47] W. X. Zhao *et al.*, *A survey of large language models*, Mar. 11, 2025. DOI: 10.48550/arXiv.2303.18223. arXiv: 2303.18223[cs]. [Online]. Available: http://arxiv.org/abs/2303.18223 (visited on 03/17/2025).

[48] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, *A survey on large language models for code generation*, Nov. 10, 2024. DOI: 10.48550/arXiv.2406.00515. arXiv: 2406.00515[cs]. [Online]. Available: http://arxiv.org/abs/2406.00515 (visited on 03/17/2025).

[49] W. Sun *et al.*, *Source code summarization in the era of large language models*, Jul. 9, 2024. DOI: 10.48550/arXiv.2407.07959. arXiv: 2407.07959[cs]. [Online]. Available: http://arxiv.org/abs/2407.07959 (visited on 03/17/2025).

[50] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," *ACM Trans. Softw. Eng. Methodol.*, Dec. 18, 2024, Just Accepted, ISSN: 1049-331X. DOI: 10.1145/3708522. [Online]. Available: https://dl.acm.org/doi/10.1145/3708522 (visited on 03/17/2025).

[51] OpenAI, *Chatgpt: An ai language model*, Accessed: 2025-03-16, 2024. [Online]. Available: https://openai.com/chatgpt.

[52] DeepSeek-AI *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025. [Online]. Available: https://arxiv.org/abs/2501.12948.

[53] Q. Jiang, Z. Gao, and G. E. Karniadakis, "DeepSeek vs. ChatGPT vs. claude: A comparative study for scientific computing and scientific machine learning tasks," *Theoretical and Applied Mechanics Letters*, vol. 15, no. 3, p. 100 583, May 1, 2025, ISSN: 2095-0349. DOI: 10.1016/j.taml.2025.100583. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2095034925000157 (visited on 04/23/2025).

[54] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, *CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation*, Sep. 2, 2021. DOI: 10.48550/arXiv.2109.00859. arXiv: 2109.00859[cs]. [Online]. Available: http://arxiv.org/abs/2109.00859 (visited on 04/23/2025).

[55] Z. Feng *et al.*, *CodeBERT: A pre-trained model for programming and natural languages*, Sep. 18, 2020. DOI: 10.48550/arXiv.2002.08155. arXiv: 2002.08155[cs]. [Online]. Available: http://arxiv.org/abs/2002.08155 (visited on 04/23/2025).

[56] E. Mashhadi and H. Hemmati, "Applying CodeBERT for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, ISSN: 2574-3864, May 2021, pp. 505–509. DOI: 10.1109/MSR52588.2021.00063. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9463106 (visited on 03/17/2025).

[57] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "VulRepair: A t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, New York, NY, USA: Association for Computing Machinery, Nov. 9, 2022, pp. 935–947, ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549098. [Online]. Available: https://dl.acm.org/doi/10.1145/3540250.3549098 (visited on 03/17/2025).

[58] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," in *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2021, New York, NY, USA: Association for Computing Machinery, Jun. 20, 2021, pp. 1–8, ISBN: 978-1-4503-8467-4. DOI: 10.1145/3460945.3464951. [Online]. Available: https://dl.acm.org/doi/10.1145/3460945.3464951 (visited on 04/23/2025).

[59] W. Yuan *et al.*, "CIRCLE: Continual repair across programming languages," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022, New York, NY, USA: Association for Computing Machinery, Jul. 18, 2022, pp. 678–690, ISBN: 978-1-4503-9379-9. DOI: 10.1145/3533767.3534219. [Online]. Available: https://dl.acm.org/doi/10.1145/3533767.3534219 (visited on 04/23/2025).

[60] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "RAP-gen: Retrieval-augmented patch generation with CodeT5 for automatic program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, New York, NY, USA: Association for Computing Machinery, Nov. 30, 2023, pp. 146–158, ISBN: 9798400703270. DOI: 10.1145/3611643.3616256. [Online]. Available: https://dl.acm.org/doi/10.1145/3611643.3616256 (visited on 04/23/2025).

[61] M. Chen *et al.*, *Evaluating large language models trained on code*, Jul. 14, 2021. DOI: 10.48550/arXiv.2107.03374. arXiv: 2107.03374[cs]. [Online]. Available: http://arxiv.org/abs/2107.03374 (visited on 04/23/2025).

[62] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, May 2023, pp. 2450–2462. DOI: 10.1109/ICSE48619.2023.00205. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10172590 (visited on 03/17/2025).

[63] E. A. Napoli and V. Gatteschi, "Evaluating ChatGPT for smart contracts vulnerability correction," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, ISSN: 0730-3157, Jun. 2023, pp. 1828–1833. DOI: 10.1109/COMPSAC57700.2023.00283. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10197134 (visited on 03/17/2025).

[64] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, New York, NY, USA: Association for Computing Machinery, Nov. 9, 2022, pp. 959–971, ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549101. [Online]. Available: https://dl.acm.org/doi/10.1145/3540250.3549101 (visited on 03/06/2025).

[65] S. Zhou, M. Huang, Y. Sun, and K. Li, "Evolutionary multi-objective optimization for contextual adversarial example generation," *Proc. ACM Softw. Eng.*, vol. 1, 101:2285–101:2308, FSE Jul. 12, 2024. DOI: 10.1145/3660808. [Online]. Available: https://dl.acm.org/doi/10.1145/3660808 (visited on 09/10/2024).

[66] F. Gao, Y. Wang, and K. Wang, "Discrete adversarial attack to models of code," *Proc. ACM Program. Lang.*, vol. 7, 113:172–113:195, PLDI Jun. 6, 2023. DOI: 10.1145/3591227. [Online]. Available: https://dl.acm.org/doi/10.1145/3591227 (visited on 09/10/2024).

[67] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, and H. Gall, "Adversarial robustness of deep code comment generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, 60:1–60:30, Jul. 12, 2022, ISSN: 1049-331X. DOI: 10.1145/3501256. [Online]. Available: https://doi.org/10.1145/3501256 (visited on 09/10/2024).

[68] J. Jia *et al.*, "ClawSAT: Towards both robust and accurate code models," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, ISSN: 2640-7574, Mar. 2023, pp. 212–223. DOI: 10.1109/SANER56733.2023.00029. [Online]. Available: https://ieeexplore.ieee.org/document/10123554 (visited on 09/11/2024).

[69] J. A. H. López, B. Chen, M. Saad, T. Sharma, and D. Varró, "On inter-dataset code duplication and data leakage in large language models," *IEEE Transactions on Software Engineering*, vol. 51, no. 1, pp. 192–205, Jan. 2025, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.2024.3504286. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10759822 (visited on 02/15/2025).

[70] O. Sainz, J. A. Campos, I. García-Ferrero, J. Etxaniz, O. L. d. Lacalle, and E. Agirre, *NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark*, Oct. 27, 2023. DOI: 10.48550/arXiv.2310.18018. arXiv: 2310.18018[cs]. [Online]. Available: http://arxiv.org/abs/2310.18018 (visited on 05/26/2025).

[71] C. Li and J. Flanigan, "Task contamination: Language models may not be few-shot anymore," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 16, pp. 18 471–18 480, Mar. 24, 2024, Number: 16, ISSN: 2374-3468. DOI: 10.1609/aaai.v38i16.29808. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/29808 (visited on 05/26/2025).

[72] A. Al-Kaswan, M. Izadi, and A. van Deursen, "Traces of memorisation in large language models for code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24, New York, NY, USA: Association for Computing Machinery, Apr. 12, 2024, pp. 1–12, ISBN: 9798400702174. DOI: 10.1145/3597503.3639133. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639133 (visited on 03/06/2025).

[73] R. Xu, Z. Wang, R.-Z. Fan, and P. Liu, *Benchmarking benchmark leakage in large language models*, Apr. 29, 2024. DOI: 10.48550/arXiv.2404.18824. arXiv: 2404.18824[cs]. [Online]. Available: http://arxiv.org/abs/2404.18824 (visited on 03/17/2025).

[74] Y. Li, *Estimating contamination via perplexity: Quantifying memorisation in language model evaluation*, Sep. 27, 2023. DOI: 10.48550/arXiv.2309.10677. arXiv: 2309.10677[cs]. [Online]. Available: http://arxiv.org/abs/2309.10677 (visited on 03/17/2025).

[75] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, 55:1–55:69, Dec. 23, 2023, ISSN: 1049-331X. DOI: 10.1145/3631974. [Online]. Available: https://doi.org/10.1145/3631974 (visited on 02/15/2025).

[76] J. S. Bradbury and R. More, *Addressing data leakage in HumanEval using combinatorial test design*, Dec. 2, 2024. DOI: 10.48550/arXiv.2412.01526. arXiv: 2412.01526[cs]. [Online]. Available: http://arxiv.org/abs/2412.01526 (visited on 05/26/2025).

[77] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, New York, NY, USA: Association for Computing Machinery, Jul. 5, 2022, pp. 1482–1493, ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510146. [Online]. Available: https://doi.org/10.1145/3510003.3510146 (visited on 09/10/2024).

[78] T. Le-Cong, D. Nguyen, B. Le, and T. Murray, *Evaluating program repair with semantic-preserving transformations: A naturalness assessment*, Feb. 19, 2024. DOI: 10.48550/arXiv.2402.11892. arXiv: 2402.11892[cs]. [Online]. Available: http://arxiv.org/abs/2402.11892 (visited on 09/17/2024).

[79] H. Ge, W. Zhong, C. Li, J. Ge, H. Hu, and B. Luo, "RobustNPR: Evaluating the robustness of neural program repair models," *Journal of Software: Evolution and Process*, vol. 36, no. 4, e2586, 2024, ISSN: 2047-7481. DOI: 10.1002/smr.2586. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2586 (visited on 09/18/2024).

[80] Q. Zhu *et al.*, "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*, ser. ESEC/FSE 2021, New York, NY, USA: Association for Computing Machinery, Aug. 18, 2021, pp. 341–353, ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3468544. [Online]. Available: https://doi.org/10.1145/3468264.3468544 (visited on 03/05/2025).

[81] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, New York, NY, USA: Association for Computing Machinery, Jul. 18, 2020, pp. 101–114, ISBN: 978-1-4503-8008-9. DOI: 10.1145/3395363.3397369. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397369 (visited on 03/05/2025).

[82] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, New York, NY, USA: Association for Computing Machinery, Jul. 5, 2022, pp. 1506–1518, ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510222. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510222 (visited on 03/06/2025).

[83] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "SelfAPR: Self-supervised program repair with test execution diagnostics," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, New York, NY, USA: Association for Computing Machinery, Jan. 5, 2023, pp. 1–13, ISBN: 978-1-4503-9475-8. DOI: 10.1145/3551349.3556926. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3556926 (visited on 03/06/2025).

[84] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, *GPT-neo: Large scale autoregressive language modeling with meshtensorflow*, version v1.1.1, Oct. 6, 2021. DOI: 10.5281/zenodo.5551208. [Online]. Available: https://zenodo.org/records/5551208 (visited on 03/06/2025).

[85] B. Workshop *et al.*, *BLOOM: A 176b-parameter open-access multilingual language model*, Jun. 27, 2023. DOI: 10.48550/arXiv.2211.05100. arXiv: 2211.05100[cs]. [Online]. Available: http://arxiv.org/abs/2211.05100 (visited on 03/06/2025).

[86] B. Rozière *et al.*, *Code llama: Open foundation models for code*, Jan. 31, 2024. DOI: 10.48550/arXiv.2308.12950. arXiv: 2308.12950[cs]. [Online]. Available: http://arxiv.org/abs/2308.12950 (visited on 03/06/2025).

[87] F. Li, J. Jiang, J. Sun, and H. Zhang, *Evaluating the generalizability of LLMs in automated program repair*, Mar. 12, 2025. DOI: 10.48550/arXiv.2503.09217. arXiv: 2503.09217[cs]. [Online]. Available: http://arxiv.org/abs/2503.09217 (visited on 03/16/2025).

[88] P. Xue *et al.*, *Exploring and lifting the robustness of LLM-powered automated program repair with metamorphic testing*, Oct. 10, 2024. DOI: 10.48550/arXiv.2410.07516. arXiv: 2410.07516. [Online]. Available: http://arxiv.org/abs/2410.07516 (visited on 11/11/2024).

[89] A. Silva and M. Monperrus, "Repairbench: Leaderboard of frontier models for program repair," arXiv, Tech. Rep. 2409.18952, 2024. [Online]. Available: https://arxiv.org/abs/2409.18952.

[90] N. Smith, D. van Bruggen, and F. Tomassetti, *JavaParser Visited*. Leanpub, 2023, https://leanpub.com/javaparservisited.

[91] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. DOI: 10.1002/spe.2346. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document.

[92] M. R. I. Rabin and M. A. Alipour, "ProgramTransformer: A tool for generating semantically equivalent transformed programs," *Software Impacts*, vol. 14, p. 100 429, Dec. 1, 2022, ISSN: 2665-9638. DOI: 10.1016/j.simpa.2022.100429. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2665963822001130 (visited on 03/12/2025).

[93] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22, Red Hook, NY, USA: Curran Associates Inc., Nov. 28, 2022, pp. 24 824–24 837, ISBN: 978-1-71387-108-8. (visited on 05/13/2025).

[94] Hugging Face, *Hugging face: The ai community building the future*, Accessed: 16-Mar-2025, 2025. [Online]. Available: https://huggingface.co.

[95] W. Kwon *et al.*, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[96] E. Nijkamp *et al.*, *CodeGen: An open large language model for code with multi-turn program synthesis*, Feb. 27, 2023. DOI: 10.48550/arXiv.2203.13474. arXiv: 2203.13474[cs]. [Online]. Available: http://arxiv.org/abs/2203.13474 (visited on 03/16/2025).

[97] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014, ISSN: 1099-1689. DOI: 10.1002/stvr.1486. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1486 (visited on 04/04/2025).

[98] K. J. Preacher and N. E. Briggs, *Calculation for fisher's exact test*, 2015.

[99] T. S. community, *fisher$_e$xact|SciPyv*1.15.3*Manual*. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.fisher_exact.html.

[100] J. Perktold, S. Seabold, J. Taylor, statsmodels-developers., *ANOVA - statsmodels 0.14.4*. [Online]. Available: https://www.statsmodels.org/stable/anova.html.

[101] J. Hauke and T. Kossowski, "Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data," *Quaestiones geographicae*, vol. 30, no. 2, pp. 87–93, 2011.

[102] T. S. community, *spearmanr — SciPy v1.15.3 Manual*. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html.

[103] T. A. S. Foundation, *Apache/commons-lang: Apache commons lang*. [Online]. Available: https://github.com/apache/commons-lang.

[104] T. A. S. Foundation, *Extendedmessageformat (apache commons lang 3.17.0 api)*. [Online]. Available: https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/text/ExtendedMessageFormat.html.

[105] Y. Yang, H. Fan, C. Lin, Q. Li, Z. Zhao, and C. Shen, "Exploiting the adversarial example vulnerability of transfer learning of source code," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 5880–5894, 2024, Conference Name: IEEE Transactions on Information Forensics and Security, ISSN: 1556-6021. DOI: 10.1109/TIFS.2024.3402153. [Online]. Available: https://ieeexplore.ieee.org/document/10531252 (visited on 09/18/2024).

[106] W. Zhang, S. Guo, H. Zhang, Y. Sui, Y. Xue, and Y. Xu, "Challenging machine learning-based clone detectors via semantic-preserving code transformations," *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3052–3070, May 2023, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.2023.3240118. [Online]. Available: https://ieeexplore.ieee.org/document/10028657 (visited on 09/12/2024).

[107] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 1, pp. 1169–1176, Apr. 3, 2020, Number: 01, ISSN: 2374-3468. DOI: 10.1609/aaai.v34i01.5469. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5469 (visited on 09/12/2024).

[108] B. Berabi, J. He, V. Raychev, and M. Vechev, "TFix: Learning to fix coding errors with a text-to-text transformer," in *Proceedings of the 38th International Conference on Machine Learning*, ISSN: 2640-3498, PMLR, Jul. 1, 2021, pp. 780–791. [Online]. Available: https://proceedings.mlr.press/v139/berabi21a.html (visited on 05/14/2025).

[109] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, Jan. 2023, ISSN: 1939-3520. DOI: 10.1109/TSE.2022.3147265. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9699412 (visited on 05/14/2025).

[110] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin, "SeqTrans: Automatic vulnerability fix via sequence to sequence learning," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 564–585, Feb. 2023, ISSN: 1939-3520. DOI: 10.1109/TSE.2022.3156637. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9729554 (visited on 05/14/2025).

[111] C. S. Xia, Y. Ding, and L. Zhang, "Revisiting the plastic surgery hypothesis via large language models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 11, 2023, pp. 522–534. DOI: 10.1109/ASE56229.2023.00047. arXiv: 2303.10494[cs]. [Online]. Available: http://arxiv.org/abs/2303.10494 (visited on 05/14/2025).

[112] K. Jain and C. L. Goues, *TestForge: Feedback-driven, agentic test suite generation*, Mar. 18, 2025. DOI: 10.48550/arXiv.2503.14713. arXiv: 2503.14713[cs]. [Online]. Available: http://arxiv.org/abs/2503.14713 (visited on 06/13/2025).

[113] N. Mündler, M. N. Müller, J. He, and M. Vechev, *SWT-bench: Testing and validating real-world bug-fixes with code agents*, Feb. 7, 2025. DOI: 10.48550/arXiv.2406.12952. arXiv: 2406.12952[cs]. [Online]. Available: http://arxiv.org/abs/2406.12952 (visited on 06/13/2025).

[114] L. Lemner, L. Wahlgren, G. Gay, N. Mohammadiha, J. Liu, and J. Wennerberg, *Exploring the integration of large language models in industrial test maintenance processes*, Sep. 10, 2024. DOI: 10.48550/arXiv.2409.06416. arXiv: 2409.06416[cs]. [Online]. Available: http://arxiv.org/abs/2409.06416 (visited on 06/13/2025).

[115] K. Li and Y. Yuan, *Large language models as test case generators: Performance evaluation and enhancement*, Apr. 20, 2024. DOI: 10.48550/arXiv.2404.13340. arXiv: 2404.13340[cs]. [Online]. Available: http://arxiv.org/abs/2404.13340 (visited on 06/13/2025).

[116] T. S. B.B, *TIOBE Index - TIOBE*. [Online]. Available: https://www.tiobe.com/tiobe-index/.