HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Enhancing Automated Program Repair with Additional Test Case Generation

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von:	Marwin Linke
geboren am:	04.05.2003
geboren in:	Rüdersdorf bei Berlin
Gutachter/innen:	Prof. Lars Grunske
	Dr. Thomas Vogel

eingereicht am: .... verteidigt am: ....

Abstract. Over the last decades, automated program repair has emerged as one of the most prominent research areas within automatically debugging defects. Despite newer advances in *learning-based* and *pattern-based repairs*, former *search-based strategies* are still widely adopted and follow a clear structure, wherein patch candidates are iteratively *generated and validated.* Before the first generation of candidates, *fault localization* identifies faulty code lines based on an existing test suite. Throughout the repair process, each iteration validates the candidates, enabling it to refine the population via *genetic programming.*

Both fault localization and validation rely heavily on *test cases*, which can be generated automatically. In principle, more test cases should help fault localization pinpoint the location of the defect, while validation should become more precise, enhancing the genetic algorithm. Both aspects should result in better patches and higher repair success rates. However, prior studies have shown, that this is not necessarily the case—this thesis aims to break down, how exactly additional test cases affect search-based program repairs. For that, GENPROG is used from the novel program repair framework FixKit to evaluate eight bugs in total, including four example programs and one real-world defect, under various configurations.

In particular, the effects on fault localization and validation are evaluated *separately*, demonstrating where additional tests yield the most significant change. This approach reveals that increasing the number of test cases during validation effectively eliminates *overfitted patches*, whereas test-enhanced fault localization provides no noticeable improvements. Modifying the way code locations from fault localization are utilized results in *better success rates* for incorporating large test suites, which otherwise produce ineffective repairs. While the addition of tests leads to only a slight increase in repair time for fault localization, the impact is considerably more pronounced when incorporated iteratively for patch validation.

Contents

1 Introduction

Fixing bugs in software is often an intricate task, including *finding*, *analyzing* and *solving* the failure without introducing unwanted behavior. Therefore, it relies on many parts to function with many parameters to refine. As for automated *search-based repairs*, they follow similar steps a human would take when *debugging* a fault. They begin by defining what makes a bug faulty, outlining which behavior is expected and which causes problems. This separation is achieved through specifying a *test suite* and *oracle*. For system tests, generating test cases is often trivial—failing and passing inputs can be used, the creation of a sufficient oracle is generally more challenging though. Once an appropriate test suite and oracle is in place, fault localization approaches can find code lines which have a high likelihood of contributing to the faulty behavior. Then, based on these code lines the program is mutated, which creates an alternated version of it, also called a *patch candidate*. By validating the candidate against the test suite, it can be inferred whether the candidate fixed the fault (whenever all test cases pass). However, fixes are only specified as valid based on the provided test suites and oracle, which often results in overfitted patches. They only satisfy the defined constraints, possibly generating faulty, incomplete or even harmful code. Several studies [20, 23, 31, 32] concluded that *generate-and-validate approaches* like GENPROG [29] produce significant amounts of overfitted patches, from which the genetic refinement suffers the most. GENPROG uses a fitness score as the heuristic function to guide the generation of new patch candidates, improving them iteratively by selecting the ones with the best fitness and producing crossover candidates. Whenever an overfitted patch occurs, the validation assigns it a high fitness score, thus leading to the selection and refinement of further overfitted patches. If the fitness score of a patch candidate reaches 1, the repair is stopped and the candidate is returned as an optimal patch.

Throughout the entire process of generating and validating patches, a test suite is indispensable for the repair, but the amount of test cases seems to play a secondary

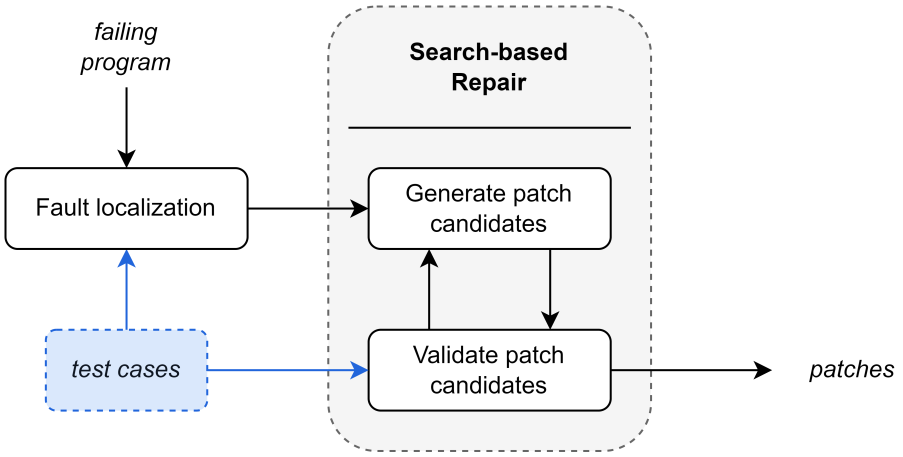


Figure 1: Conceptual structure of generate-and-validate repairs, highlighting the importance of test cases.

role. As prior studies [13,34] have shown, additional test cases do not strongly correlate with the repair's success rate but are beneficial in reducing overfitted patches [31,32]. Nonetheless, due to the incomplete nature of test suites, automated repairs struggle to find generalized solutions [34]. Although these studies have shown no clear correlation between additional test cases and repair success rates, it is not entirely obvious why.

During this bachelor's thesis, the automated program repair collection FixKit [25] was extended to allow an easy evaluation of the subjects implemented in Tests4Py [24]. With automatically generated test cases through a grammar-based fuzzer by ISLa [27], different repair configurations can be evaluated. In particular, the evaluation utilizes test cases separately in the fault localization and validation portions of the repair process. This allows for a more detailed analysis to observe whether the implemented fault localization techniques can be enhanced and result in higher repair success rates, as well as to confirm whether more test cases in the validation process eliminate overfitted patches. In total, this thesis provides the following contributions:

Evaluation of FixKit. The novel repair framework FixKit [25] was evaluated using eight bugs from four toy subjects and one real-world defect, all implemented in Tests4Py [24]. Since both tools may lack verification for validity, this thesis also examines whether their combination can be effective in evaluating and analyzing automated program repair.

Differences in Effectiveness. To directly demonstrate the impact of additional test cases on different stages of the repair process, this evaluation separates fault localization and validation in terms of the tests used.

Modification of Code Locations. Initial evaluations indicated a decline in repair performance with larger test suites, partially due to how suggestions from fault localization were applied. By modifying how FixKit utilizes these suggestions, this evaluation provides further insights into their impact on repair effectiveness.

Evaluation Pipeline. An evaluation pipeline was implemented following the modular structure of FixKit. This design facilitates further experimentation with various configurations, such as different numbers of test cases or alternative repair approaches.

Note. In this paper, the term *test cases* is used interchangeably with *inputs*, since the generated inputs can directly be used as system test cases. To define the correctness of inputs, an oracle is provided, which tells the program whether an input classifies as *failing* or *passing*. Moreover, *tests* is used as a shortened form of test cases, but holds the same meaning.

Structure. Section 2 presents the background for this thesis, explaining what approaches and tools are relevant for the evaluation. In Section 3, the types of automated program repair are described and what challenges are given with *search-based repairs* concerning the use of test suites. Section 4 describes the implementation of the *evaluation pipeline* along the repair process, going into detail where and how test cases might affect the repair. Moreover, it explains why the suggestions of fault localization led to ineffective repairs and how *modifiers* can change this behavior. Next, Section 5 elaborates on how the produced patches are evaluated, showing the concrete setup in Subsection 5.2. The evaluation results are presented in Section 6, answering three research questions, and discussed in Section 7. The threats to validity can be found in Section 8, whereas Section 9 concludes this thesis by giving a summary and an overview of possible future work.

2 Background

This section provides background on the approaches and tools relevant to the implementation and evaluation of this thesis.

Automated Program Repair. This thesis mainly builds on top of the *repair framework* named FixKit [25], which implements the search-based repair approaches GENPROG [29], MUTREPAIR [5], KALI [21], CARDUMEN [16] and AE [28] in Python. The core functionality, it being genetic program repair guided by a heuristic function, is shared among the different approaches. Its modular structure facilitates the extension of further approaches regarding the repair method, fault localization, fitness metric and validation engine. These parts of the repair are interchangeable, enabling it to evaluate various configurations. It comes with direct support for using the *testing framework* Tests4Py [24], which allows the validation engine to calculate the fitness scores without further implementations.

The evaluation was limited to one repair approach due to the high number of configurations required for notable insights. The choice fell on GENPROG [29] since it performed the best in early experiments and supports the best integration with system test cases out of the box, while other repair approaches in FixKit would have needed further implementations to function reliably. GENPROG was one of the primary approaches to lead *genetic search-based repairs* in 2011, introducing a fitness heuristic to improve patches over multiple iterations. Subsection 3.1 gives an overview of the current types of program repair, while Subsection 4.2 details how FixKit implements GENPROG.

Testing Framework. The testing framework Tests4Py [24] is directly supported in FixKit [25], allowing for easy integration and evaluation of the subjects implemented in Tests4Py. The framework is derived from BugsInPy [30] and tries to address the limitations of inadequate system oracles and sparse unit tests. Every subject features faulty versions of a program, together with predefined unit tests, system tests, oracles and grammars. In total, four example programs (toy subjects) are included, which can easily be debugged and may give conceptual insights, as well as seven real applications. Due to complications with generating inputs and inconclusive repair results, only the toy subjects and two bugs from the real-world program Pysnooper [22] could be evaluated.

Moreover, Tests4Py provides functional grammars and oracles not suitable for grammar fuzzing due to a verbose structure. `debugging-benchmark` [6] is a tool, which was directly developed to solve this issue by acting as a harness for Tests4Py. It implements the subjects with simpler grammars and hand-defined initial inputs, which were important for using approaches like Avicenna [7] to generate inputs.

Grammar Fuzzing. In this thesis, a grammar fuzzer is used to generate inputs for the test suites. The specific grammar fuzzer is provided by ISLa [27], which reimplements the code from the Fuzzing Book [35] with optimizations such as using custom derivation tree objects and an iterative traversal, which is supposed to increase performance. In detail, the grammar fuzzer builds a derivation tree, from which the leaves, that represent the terminal symbols, can be concatenated and read as a fuzzed string. To build the tree, each non-terminal acts as a node which can be expanded according to the possible expansions given through the grammar. A simple strategy would be to randomly choose an expansion for every node, but this might lead to long recursive loops, which are unnecessary.

The grammar fuzzer follows a more effective strategy, which chooses the expansions based on a cost function. The cost for an expansion is calculated by the sum of the symbol cost of each non-terminal in the expansion, while the symbol cost is calculated from the minimum cost of all possible expansions. Whenever a symbol that was already seen is encountered again, the cost is set automatically to infinite. This is a recursive way to find the expansion with the lowest cost, which collapses the tree. With that approach, the number of non-terminals will always decrease if the grammar is valid, eventually ending in only terminal symbols. However, to generate diverse results, the grammar fuzzer utilizes a three-way strategy:

1. The tree is expanded with a maximum cost function until a certain minimum number of non-terminals are present.
2. The tree is randomly expanded until a certain maximum number of non-terminals are reached.
3. The tree is closed up with the before-described minimum cost expansion.

This forms an efficient way to fuzz inputs from a grammar. A more detailed explanation can be read in the Fuzzing Book[1] [35].

Automated Bug Diagnoses. Instead of using a grammar fuzzer, this thesis first aimed to use a novel approach named Avicenna [7] to generate inputs. Avicenna combines the ideas of Alhazen [11] and ISLearn [27] to produce automated bug diagnoses, which are compromised of failing constraints based on a defined grammar.

[1]`https://www.fuzzingbook.org/html/GrammarFuzzer.html`, 20.02.2025

While this is particularly interesting as a way to describe bugs in a language (namely the ISLa [27] language), which can be read by humans and processed by computers, this also provides an opportunity to generate inputs, which are directly tailored to the input constraints of a bug. Tools like the `ISLaSolver` [27] can generate new inputs from the produced bug diagnoses, having better chances of finding faulty inputs. However, after some experiments, issues led to the choice of using the grammar fuzzer as a more simple and reliable method for generating inputs. Subsection 4.1 explains some methods of generating inputs in more detail.

3 Related Work

3.1 Types of Program Repair

Automated program repair, in its most fundamental form, involves receiving a defective program as input and generating a patch that represents a corrected version of that program. To accomplish this, the repair usually infers information from other techniques, such as fault localization. As outlined in the introduction, *search-based repairs* work on the premise of *finding* the location of the fault, *generating* a patch candidate and *validating* it against a test suite. Search-based repairs often leverage the capabilities of *genetic programming* by utilizing heuristic metrics to evaluate patch quality and iteratively refine candidates. Since the introduction of GENPROG [29] in 2011, which popularized this strategy, numerous other repair types have emerged.

As an alternative to search-based repairs, *semantic-based repairs* like SEMFIX [18] arose in 2013. While these approaches also use fault localization to rank program statements based on their suspiciousness scores, they employ *static symbolic execution* with the ranked statements and a test suite as inputs. Symbolic execution is able to extract repair constraints, which can be transformed into a repair by solving them via program synthesis—the Z3 SMT solver [4] is used in the case of SEMFIX. When evaluated against GENPROG [29], the original study on SEMFIX [18] reported equal or improved success rates.

Around the same time, *pattern-based repairs* like PAR [12] shifted the focus to human-readable patch generation. Unlike former approaches that directly mutated code lines, often leading to redundant or difficult-to-interpret modifications, pattern-based repairs use code templates to alternate the program's behavior. In the case of PAR, fault localization is employed as the first step, after which an analysis process scans the program's abstract syntax tree (AST), analyzing the identified fault location and its adjacent locations. If the scanned code lines match a template, PAR rewrites the AST based on the predefined editing script in the templates. Similarly to GENPROG [29], the patch candidates are validated and used for an evolutionary feedback loop. While PAR can incorporate *human-written* templates and generate more readable patches, it is evaluated to only perform marginally better than GENPROG [12].

More Recently, a large quantity of *learning-based strategies* have been proposed that use machine learning to enhance automated repairs. For example, DeepFix [9] was introduced in 2017 and directly repairs faults by predicting patch candidates through deep learning approaches, similarly, DeepRepair [33] and Dear [15] are newer approaches that enhance the use of deep learning in automated program repair. The use of machine learning has been a focal point of research over the past years, especially with the rise of large language models (LLMs) that commonly utilize transformer neural networks. Fittingly, in 2021 TFix [3] was proposed, which directly leverages the powerful tranformer-based models pretrained on natural language to generate patches text-to-text.

Despite many new advances in repair approaches and impressive enhancements, each comes with its own set of challenges. As for search-based repairs, they still provide versatile functionality that works with minimal input data and offers a clear structure. This allows researchers to understand the fundamentals of generating patches, how to approach challenges and find solutions across multiple repair types. For most forms of program repair, the test suites either play a superficial role or do not benefit from a larger size. Many studies looked into this behavior for search-based repairs, as explained in the next subsections.

3.2 Effectiveness of Test Suites

In their study, X. Kong et al. [13] found no clear correlation between increasing the test suite size and the success rates of genetic search-based repair techniques. However, their analysis of brute-force and adaptive repair techniques—alternative approaches to the genetic search—revealed more notable trends. Specifically, as the number of failing tests increased, the success rates of these techniques showed a declining trend. In contrast, the addition of passing tests resulted in either stable success rates or only a slight decrease.

Their theory on the impact of larger test suites is described as twofold: (1) Larger test suites inherently increase the difficulty of all test cases passing the validation, thereby lowering the reported success rates. This is a reasonable assumption, as larger test suites introduce additional constraints the repair must satisfy. However, evaluating only the tool-reported success rates does not capture patch quality. As explained in the following subsection, while smaller test suites may lead to higher success rates, they are also more susceptible to overfitting. Notably, their findings apply only to tool-reported success rates, which included overfitted patches. In contrast, no clear relationship between test suite size and the actual repair success rate (when assessed by humans) was observed across all evaluated techniques.

(2) Another observation in their study is that an increase in the number of passing test cases enhances fault localization, thus overcoming the stricter requirements for patches to pass due to the larger test suite, resulting in a stable trend. However, a separate study by Y. Lei et al. [14] found no strong correlation between test suite size and the effectiveness of fault localization performance. Instead, they found a positive impact on fault localization coming from passing tests that do not execute the faulty statements and failing tests in general, while passing tests that exercise the faulty statements harm effectiveness. This partially contradicts the earlier findings

of X. Kong et al. [13], where failing tests did not appear to contribute positively to localizing and repairing the fault.

Accuracy of Fault Localization. A study [2] on the accuracy of spectrum-based fault localization reported a detailed analysis of the effect of failed and passed runs: R. Abreu et al. observed that only a few failed runs (around 6) are enough to reach near-optimal diagnostic performance, with more failed runs below that threshold only accumulating the performance. However, the effect of including more passed runs was observed to be unpredictable, either improving or degrading diagnostic performance. Including more than about 20 passed runs had little effect on the accuracy.

The statistical metric utilized in fault localization plays a crucial role in the effectiveness and accuracy of the results. While Ochiai [1], the metric used for the evaluation of this thesis, reported to be effective for fault localization [1, 2], often outperforming other options, the work of Y. Qi et al. [19] proposed a new research direction: Viewing the effectiveness of fault localization directly through automated program repair. They found two critical observations: (1) fault localization techniques, which performed well in prior studies, did not have good performance for automatically repairing faults. (2) In contrast to other studies, Jaccard [2] seems to perform at least as well as other techniques, with partially larger effects regarding improved effectiveness and accuracy.

3.3 Overfitting

When program repair relies on test suites as the primary criterion for generating patches, there is a significant risk of producing overfitted patches—patches that satisfy the given test cases but fail to address the underlying fault. Multiple evaluations [31, 32] have demonstrated that (using the setups of state-of-the-art techniques as of 2017) over half of the generated patches were incorrect despite being treated as valid by the repair.

To mitigate the occurrence of overfitting, approaches such as Opad [32] have been developed. Opad generates additional test cases specifically to assess whether a patch overfits the fault. These additional test cases are only applied once a candidate patch has been identified as valid by the repair. To filter out overfitted patches effectively, Opad implemented a custom metric named O-measure. The larger test suite in this context serves as an additional validation layer to eliminate overfitted patches, rather than directly influencing the genetic evolution process during repair. With Opad in place, J. Yang et al. [32] reported successfully filtering out 75.2% of overfitted patches in their evaluation.

3.4 Inconsistent Causes

Despite numerous studies investigating the effects of larger test suites and overfitted patches, it is not easy to pinpoint the exact cause of these observations. The reason often lies in the large range of possible parameters, which led to less comparable results among studies. For example, fault localization studies may provide contrasting results due to differences in possible metrics, techniques, amount of test runs, program sizes, and choice of subjects. Moreover, techniques that appear *non-deterministic* like genetic search-based repairs are very vulnerable to the randomness of mutations and often report varying results for slight changes in parameters. In this thesis, the effects on fault localization and validation are evaluated separately to gain better insights into the exact causes, which lead to either positive or negative trends with additional test cases.

4 Improving Repair with Additional Tests

This section explains the integration of additional test cases into the repair process, detailing the implementation and selected parameters. Subsection 4.1 provides a motivating example, helping to understand how subjects and inputs are structured, while also presenting different methods for input generation. The repair process is described in Subsection 4.2, highlighting the components that depend on the test suite as well as other important parameters. Finally, Subsection 4.3 explores how suggestions from fault localization can be modified to incorporate larger test suites more effectively into the repair process.

4.1 Generation of Test Cases

In general, search-based repair techniques require failing and passing test cases. To illustrate how test cases are generated, we present a motivating example about the infamous subject `middle_2`. The task of `middle_2` is to return the middle value of three integers provided as parameter inputs. An easy way to solve this is by sorting the three integers and returning the value of the second one. However, the implementation of `middle_2` in Tests4Py [24] intentionally includes a defect. As presented in the evaluation results, FixKit [25] can repair the fault of `middle_2`, simply by utilizing test cases, the source code of the program and an oracle, which is necessary to declare inputs as failing or passing. Figure 2 depicts a comparison of the source code of `middle_2` with one being the faulty and the other one being the patched version. To easily recognize faulty inputs of `middle_2`, one can check if the following statement holds true for inputs of the form `(x, y, z)`:

$$\mathtt{y} < \mathtt{x} < \mathtt{z} \tag{1}$$

FixKit [25] handles these test cases by loading the inputs as paths of individual text files, with each argument being separated by spaces. This means failing inputs such as `1 0 2` need to be written into a text file which can be saved as `"tmp/middle_2/failing_test_0"` in a temporary directory. While not essential for the repair, many modern input generation methods need additional information such as grammars to produce them effectively. The following descriptions outline the importance of grammars and oracles for generating inputs:

```python
1  def middle(x, y, z):
2  if y < z:
3      if x < y:
4          return y
5      elif x < z:
6          return y
7  else:
8      if x > y:
9          return y
10     elif x > z:
11         return x
12 return z
```

(a) Faulty implementation at line 6.

```python
1  def middle(x, y, z):
2  if y < z:
3      if x < y:
4          return y
5      elif x < z:
6          return x
7  else:
8      if x > y:
9          return y
10     elif x > z:
11         return x
12 return z
```

(b) Patched version.

Figure 2: Python source code of `middle_2`.

Grammar. A simple way of gathering test cases is fuzzing, which was introduced in 1990 [17]. Fuzzing is the act of generating inputs at random until they fail a program, thus a faulty input is found. General fuzzing can often be enhanced by incorporating grammars, which directly define the structure of valid inputs. For example, instead of fuzzing random strings for `middle_2` and hoping that it generates three integers and two spaces in between purely by chance, a grammar can specify that the inputs are exactly structured how they should be syntactically. Therefore, only valid inputs (not necessarily passing) will be fuzzed. Figure 3 shows the code used to define the grammar of `middle_2`.

Oracle. Additionally, a failing condition must be specified to let the fuzzer know what exactly counts as a failing input. In the case of `middle_2`, the oracle asserts that the returned value is the second value of the sorted arguments. Once this assertion fails, the input is classified as failing and otherwise as passing.

```python
middle_grammar: Grammar = {
    "<start>": ["<x> <y> <z>"],
    "<x>": ["<integer>"],
    "<y>": ["<integer>"],
    "<z>": ["<integer>"],
    "<integer>": ["<digit>", "<digit><integer>"],
    "<digit>": [str(num) for num in range(1, 10)],
}
```

Figure 3: Python source code of the grammar of `middle_2`.

4.1.1 Subjects

All subjects evaluated in this thesis are provided by Tests4Py [24]. The testing framework presents subjects by their name and `bug_id`. While the name indicates the program in general, the `bug_id` refers to a certain variation of the program in which only one bug is present. Thus, `middle_1` and `middle_2` are slightly different implementations of the same program and need alternate fixes.

This paper either refers to a subject by stating its `bug_id` directly behind the name of the subject, the way it is for `middle_2` for example, or by stating the subject name on its own if the `bug_id` is not relevant or if there is only one bug for a subject. As for the generation of test cases, subjects like `middle_1` and `middle_2` require the same inputs and therefore do not need to be differentiated in this context.

4.1.2 Generation Methods

There are many different ways of generating inputs. The next paragraphs highlight which methods were considered for the evaluation and how they differ.

Tests4Py. Tests4Py [24] itself implements 10 predefined failing and passing inputs for each subject, providing a baseline of test cases. These inputs can be found in the `"tmp/{subject}"` directory for each subject, respectively.

Grammar-based Fuzzer. Alternatively, a grammar-based fuzzer can generate additional test cases. Using a corresponding grammar and oracle, which are implemented in `debugging-benchmark` [6], a wrapper for Tests4py [24], the fuzzer can generate indefinitely many test cases given enough time. Table 1 provides an overview of how many iterations it took the fuzzer to generate at least 250 failing and 250 passing inputs. Moreover, the time needed to generate those test cases is fairly low, moving in the territory of seconds for toy subjects such as `middle_2`. The results in the table and generated test cases for the evaluation were produced by the `GrammarFuzzer` from ISLa [27]. The recorded times were measured on a local machine during early experiments.

Avicenna. AVICENNA [7] is a novel approach from 2023, which was considered as a method for test case generation due to it having prior connections to Tests4Py [24], showing the ability to generate bug diagnoses for the subjects. While bug diagnoses are primarily a method to explain bugs in a semantic way, which can be understood by humans as well as computers, this very property also allows it to effectively find failing inputs. Using specified grammars, initial inputs and oracles, which are all available from `debugging-benchmark` [6] and Tests4Py [24], AVICENNA could be employed to generate additional test cases. It utilizes a feedback loop to refine a hypothesis by determining logical characteristics over input elements, which are extracted from the grammar. For example, AVICENNA can conclude that for every input of `middle`, it

Table 1: The number of generated test cases through a grammar-based fuzzer in relation to the total number of iterations and measured time.

Subject	#Failing inputs	#Passing inputs	Iterations	Time
`calculator`	250	250	2513	< 3 s
`middle`	250	250	1638	< 3 s
`markup`	250	250	8348	~ 30 s
`expression`	250	250	1081	< 3 s
`pysnooper_2`	250	250	2151	~ 3 m
`pysnooper_3`	250	250	1047	~ 2 m

must follow the equation shown beforehand in 1. Tools like the `ISLaSolver` [27] can append on the functionality and *solve* the produced diagnosis from AVICENNA.

However, the generated inputs fall short of using a grammar-based fuzzer, because the faults implemented in Tests4Py are reasonable to find at random given the structure of the grammar. Due to the overhead of generating inputs through a diagnosis, the `ISLaSolver` takes significantly longer, often requiring up to 30 minutes per subject to generate the same amount of test cases as shown in Figure 1. Moreover, some subjects cannot reliably generate failing inputs in high quantities. As for passing inputs, they can be generated by negating the formula of the diagnosis. This also did not work reliably for most subjects. Due to the missing reliability and high dependency on the `ISLaSolver`, this evaluation did not generate test cases through AVICENNA. Despite falling short, we still want to highlight some benefits:

AVICENNA uses initial inputs to build a hypothesis over multiple iterations, meaning the resulting inputs are only related to one bug. The ability to distinguish between multiple bugs might be beneficial for automated program repair, allowing the genetic refinement to focus on the same objective. However, we deemed this advantage unnecessary for this thesis, since the subjects from Tests4Py aim to implement only one bug at a time. Nonetheless, if better methods are introduced to *solve* the diagnoses provided by AVICENNA, it might be able to generate test cases quicker and more reliably in the future.

4.2 Repair

The repair approaches implemented in FixKit [25] generally follow the common *generate-and-validate* strategy, while leveraging the power of genetic programming to iteratively improve patch candidates. They incorporate a fitness function, which evaluates the quality of a patch and decides which candidates stay in the current population for the next iteration. Figure 4 presents the repair process in detail, demonstrating how GENPROG [29] operates and behaves within FixKit.

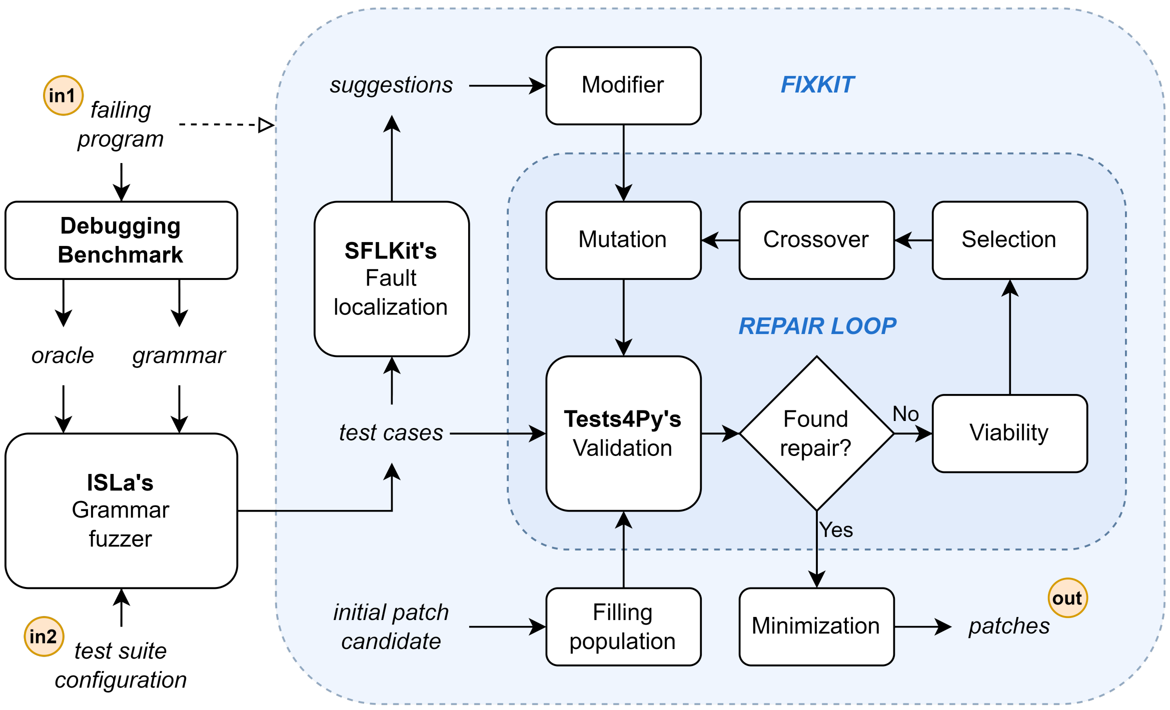


Figure 4: Structure of FixKit's [25] repair process.

4.2.1 Repair Process

FixKit's [25] repair process consists of three distinct parts: *fault localization*, *patch generation* and *patch validation*. While localizing the fault is only part of the preparation, thus being executed once, the generation and validation form a repair loop. One creates new patch candidates through crossovers and mutations, while the other one validates these new candidates by assigning fitness scores, which are relevant for selecting patch candidates in the next generation.

Fault Localization. Before localizing the fault, FixKit [25] prepares the repair process by parsing the source files of the faulty program through a statement finder, which maps code lines to an identifier number. This is later used to store the output of fault localization in pairs of identifiers and weights (called suggestions).

Fault localization techniques identify code lines that are likely contributors to a program's fault. These methods typically rely on statistical metrics derived from a given test suite to rank code lines by their level of *suspiciousness*. Code lines that are executed more frequently during failing tests and less frequently during passing tests are inferred to have a higher likelihood of causing the fault. After ranking the suspiciousness of code lines, the most likely faulty ones are mutated or swapped. This introduces a new patch candidate, which causes an alternative behavior of the program. To guide the candidates in a beneficial direction, GenProg [29] only considers other code lines present in the program as valid alternatives, this follows the *competent programmer's hypothesis* [8]. The idea behind it is that bugs in real-world programs only deviate slightly from the intended code and therefore can be resolved by simple mutations. This greatly reduces the search space required for finding an appropriate patch candidate which fixes the fault. However, this behavior also limits the ability to repair faults in smaller programs or in those with unique, non-repetitive code lines.

FixKit [25] employs the statistical fault localization workbench SFLKit [26] to find the location of faulty code lines with OCHIAI [1] as the statistical metric because studies observed it to be very effective among other metrics [1,2]. Commonly used alternatives for the metric are TARANTULA [10] or JACCARD [1].

Generation of Patches. After localizing the fault, the repair process fills the population by cloning an initial candidate, which has no mutations and represents the original program. How many candidates are generated depends on the parameter for the population size. The actual number of candidates grows and shrinks during the repair process based on the following steps, which are applied to the entire population every iteration:

Viability. At the beginning of each new generation, all candidates with a fitness score of 0 are removed from the population, ensuring only useful candidates remain. The removed patches usually contain invalid program behavior.

Selection. As the next step, half of the defined population size is selected from the current population, with the fitness score as the probability of remaining in it. A higher fitness score results in a higher likelihood of staying in the population for the next generation but is not guaranteed for candidates with fitness scores below 1.

Crossover. The selected population is shuffled and ordered into pairs. From each pair of candidates, the corresponding mutations are appended crosswise and mutually, resulting in two new child candidates, doubling the population in total.

Mutation. Lastly, for each candidate, each code line receives a chance to be mutated based on the weight of the suggestion associated with that code line. For example, a mutation that is represented by `Replace(6, 10)` has the effect of replacing the code line identified as 6 with the one identified as 10. If this specific mutation is applied to `middle_1`, it repairs the fault. The way these mutations are applied by chance plays a crucial role in repairing the program, therefore this behavior was evaluated and modified, as discussed in Subsection 4.3.

Validation of Patches. At the end of each iteration, the fitness score of each candidate is updated by evaluating it with the specified failing and passing test cases for validation. FixKit [25] implements a fitness engine, which automatically employs Tests4Py [24] to evaluate all candidates in parallel and calculate the fitness scores, which are then assigned to each candidate.

Finalization. At the end of the repair process itself, be it reaching the maximum number of iterations or finding a valid candidate, the repair is finalized. *Delta debugging techniques* help to remove redundant mutations, which results in more qualitative patches. However, this should not change the success rate in any way.

4.2.2 Approaches and Parameters

Most parameters were selected to closely align with the original implementation of GENPROG [29] in FixKit [25], including a recommended population size of 40 and the use of OCHIAI [1] as the statistical metric for fault localization. However, early evaluations revealed random and unpredictable results, which were tracked down to be caused by the way fault localization suggestions are utilized. By modifying the use and general mutation probability, better results were noticed. This is discussed in detail in Subsection 4.3.

4.3 Suggestions

During the initial evaluation, an unexpected pattern in the actual repair success rate was observed. Specifically, an increase in the number of test cases often led to more unstable repairs than the baseline and even a decline in effectiveness for larger test suites in some instances. This issue was primarily traced to the way fault localization assigns suspiciousness scores and incorporates them into the patch generation process. In FixKit's [25] base implementation, each line of code has a probability of being mutated based on its assigned *weight* by the *suggestion* (and a general mutation chance). This weight corresponds to the normalized suspiciousness score calculated using the following metric:

$$Ochiai(f_o, f, p_o) = \frac{f_o}{\sqrt{f \cdot (f_o + p_o)}} \tag{2}$$

The OCHIAI [1] metric utilizes f_o and p_o to represent the number of *observed* failing and passing test cases, respectively, while f denotes the *total* number of failing test cases. This metric is computed for *each* code line based on the test cases that traverse that code line. OCHIAI is a widely adopted metric in fault localization [1, 2]—however, it comes with one flaw: It generally performs better when more total passing test cases are present than failing ones. The effects can be seen with an example.

Table 2 presents the top five suggestions from `middle_1's` fault localization using varying numbers of test cases. As expected, increasing the number of test cases improves the precision of the suggestions. Notably, the configuration with 50 failing and 50 passing tests correctly identifies code line 6 as the primary source of the fault. However, the configuration with 1 failing and 10 passing tests appears more effective for the repair process. This effectiveness arises because the lower weights assigned to the third suggestion and onward make these code lines less likely to be mutated, reducing the occurrence of ineffective mutations. Given that there can be dozens of such suggestions, those differences in mutation probabilities (e.g. 0.3 vs. 0.7) for non-faulty code lines can significantly impact the repair outcome. This difference originates from the 1:10 ratio of failing to passing test cases, as opposed to the 1:1 ratio.

Table 2: Top five suggestions from `middle_1's` fault localization for test cases (failing, passing). The first number is the identifier, the second is its associated weight.

(1, 1)	**(1, 10)**	**(10, 10)**	**(50, 50)**
5: 1.000	5: 1.000	6: 1.000	6: 1.000
6: 1.000	6: 1.000	5: 0.956	5: 0.901
1: 0.707	3: 0.500	3: 0.877	3: 0.844
2: 0.707	1: 0.302	1: 0.707	1: 0.721
3: 0.707	2: 0.302	2: 0.707	2: 0.721
...	...	...	

Intuition and Proof. This paragraph provides intuition on why the *ratio* between failing and passing test cases influences the Ochiai [1] metric and demonstrates a conceptual proof of the observed effect. When additional passing test cases are introduced without adding failing ones, p_o increases, while f and f_o remain constant. Because p_o appears in the denominator of the metric, its increase causes the value of the metric to decrease for code lines affected by these passing test cases. In FixKit [25], the metric is then normalized, so this decrease widens the gap between the faulty code line (which is less affected by passing tests) and other code lines, thereby improving fault localization for the repair when more passing tests are present. This behavior depends on the *ratio* of failing to passing tests rather than their absolute counts.

For the proof, assume there are f failing test cases and p passing test cases, such that $p = k \cdot f$, where k represents the ratio of passing to failing tests. For any specific code line, let the proportion of failing test cases traversing it be i, and the proportion of passing test cases traversing it be j. This implies $f_o = i \cdot f$ and $p_o = j \cdot p = j \cdot k \cdot f$. Substituting these into the Ochiai metric yields:

$$\text{Metric} = \frac{f_o}{\sqrt{f \cdot (f_o + p_o)}} = \frac{i \cdot f}{\sqrt{f \cdot (i \cdot f + j \cdot k \cdot f)}},$$

which can be simplified to:

$$\text{Metric} = \frac{i \cdot f}{\sqrt{f^2 \cdot (i + j \cdot k)}} = \frac{i \cdot f}{f \cdot \sqrt{i + j \cdot k}} = \frac{i}{\sqrt{i + j \cdot k}}.$$

This result shows that the metric depends only on the constants i, j, and k, confirming that the ratio of passing to failing test cases (k) plays an important role in the metric's behavior. Due to the way, FixKit utilizes the weights as the probability to apply a mutation for that specific code line, it often disables the repair to benefit from additional test cases. In practical software scenarios, the ratio of passing to failing test cases is not typically something that can be easily adjusted. Moreover, as shown in Table 2, increasing the number of failing test cases can improve the precision of repair, isolating the exact location of the fault. FixKit's default implementation of incorporating the suspiciousness score into the repair will often lead to a dependency on the ratio. To mitigate this effect and observe the behavior in the evaluation results, we included *modifiers* in FixKit, which alter the suggestions post hoc in different ways.

4.3.1 Modifiers

The suggestions generated by fault localization are stored in a list of *weighted identifiers*, where each identifier is associated with a specific code line and a weight, representing the normalized suspiciousness score on a scale from 0 to 1. The list is sorted first in descending order of weight and then in ascending order of identifier number, ensuring reproducible outcomes. Modifiers are applied directly before the repair process accesses this list, altering two aspects: which suggestions are selected from the list and how the weight influences the mutation probability, if at all. Table 3 provides an overview of how each modifier impacts the suggestions and their corresponding mutation chance based on the weight.

Table 3: Selected suggestions and their corresponding mutation chance per modifier based on the weight.

Weights	Default	TopRank	TopEqualRank	WeightedTopRank	Sigmoid
1.00	1.00	1.00	1.00	0.50	1.0000
1.00	1.00	1.00	1.00	0.50	1.0000
0.90	0.90	1.00	1.00	1.00	0.9997
0.80	0.80	–	1.00	1.00	0.5000
0.70	0.70	–	–	–	0.0045

Default. The `DefaultModifier()` does not modify the behavior of the original implementation of FixKit [25]. It returns the same list of suggestions and uses the weight as the mutation chance.

TopRank. The `TopRankModifier(k=3, threshold=0)` returns only the first `k` suggestions, which are the ones with the highest weights. The `threshold` parameter removes all suggestions with a weight at or below the specified value. The mutation chances for the selected locations are all set to 1, rather than using the weight. In theory, the `TopRankModifier()` should be more effective as it focuses solely on the highest-weighted suggestions, but it can be misleading. For example, multiple suggestions may share the same weight, often due to being traversed by the same test cases. If the first six suggestions all have a weight of 1, this modifier will only use the first three of them (with `k=3`), even though the others are also identified as faulty. To address this issue, some variations were implemented.

TopEqualRank. The `TopEqualRankModifier(k=3, threshold=0)` works similarly to the previous modifier, but returns all suggestions under the top `k` weights. This approach should better reflect the localized fault. However, many unnecessary suggestions may be included, if they share the same weight. It performs well when the suggestions are diverse, which is often the case when additional test cases are used.

WeightedTopRank. The `WeightedTopRankModifier(k=3, threshold=0)` returns the same suggestions as the previous modifier but adjusts the mutation chance based on the number of suggestions with the same weight. If n suggestions have the same weight, the mutation chance for each of these suggestions is set to $1/n$. The idea behind this is that suggestions with the same weight might originate from the same general location (e.g., one code block) because of the same number of traversed failing and passing tests going into the metric. With that modifier, this general location would receive the same total mutation chance as other parts, but divided on all code lines within it.

Sigmoid. The `SigmoidModifier(steepness=10, m=0.8)` uses a modified sigmoid-like function, which allows control over the `steepness` of the curve and the midpoint `m` at which the value of the function is always 0.5. This function pushes weights above the midpoint towards 1 and weights below it towards 0. Using the weights as mutation chances, similar to the `DefaultModifier()`, the sigmoid function may effectively isolate faulty code lines. However, due to the inconsistencies described earlier, especially when different test case ratios are used, this modifier cannot be generalized without determining the optimal midpoint for each configuration. The implemented function is as follows, where x is the weight as the input, a is the steepness and m is the midpoint:

$$f(x) = \frac{\left(\frac{x}{m}\right)^a}{\left(\frac{x}{m}\right)^a + \left(\frac{1-x}{1-m}\right)^a}$$

4.3.2 Mutation Chance

Due to the modifiers *reducing* the number of suggestions relevant to the repair process, the number of mutations also decreases equivalently. To mitigate this effect, the general mutation chance `w_mut` was increased from 0.06 to 0.2, which after some experimentation seemed like an effective value for most toy subjects. The ideal choice of the mutation chance depends on how many mutations a certain subject requires. For many, a single mutation is often enough, for example, `middle_1` can be fixed through the mutation `Replace(6,10)`. Using higher mutation chances can accelerate the process of finding the right mutation, but very high values lead to long combinations of mutations, which are rarely able to fix the program.

5 Evaluation

To evaluate the quality of produced patches, an independent test suite is employed, separate from the one used during the repair process. Using the *system tests engine* from FixKit [25], which is also utilized during validation and based on Tests4Py's [24] testing capabilities, the final fitness score is computed. Moreover, a Tests4Py report is generated, which maps all test cases to an enumeration data type called `TestResult`, which classifies each test case as either `PASSING`, `FAILING`, or `UNDEFINED`—the latter only occurs when the oracle fails to categorize a test case. Each patch candidate stores an individual instance of this report, which allows us to analyze the quality of a patch with a custom metric.

5.1 Metric

Rather than relying solely on the fitness score, a custom evaluation metric was implemented that resembles the structure and functionality of a confusion matrix. It compares the outcome of test cases to the classification of the original program:

1. `STILL PASSING`: The original passing test cases are *still passing* the patch.
2. `NOW FAILING`: The original passing test cases are *now failing* the patch.
3. `NOW PASSING`: The original failing test cases are *now passing* the patch.
4. `STILL FAILING`: The original failing test cases are *still failing* the patch.

This metric allows us to calculate more scores, such as the *precision*, *recall* and *F1 score* of patches.

$$precision = \frac{\texttt{NOW PASSING}}{\texttt{NOW PASSING} + \texttt{NOW FAILING}}$$

The *precision* expresses the advantageous changes in the patch's behavior compared to the original program, meaning a value of 1 indicates that only the faulty behavior was resolved without altering the functionality which was originally passing. In contrast, a value of 0 indicates no useful changes at all, meaning the bug still persists.

$$recall = \frac{\texttt{NOW PASSING}}{\texttt{NOW PASSING} + \texttt{STILL FAILING}}$$

The *recall* indicates how effective the patch is in repairing the fault by assessing how many test cases are now passing, which were originally failing. A value of 0 also relates to no useful changes, while a value of 1 implies that the original fault was completely resolved. However, it may include incorrect behavior in the once-correct functionality of the program.

The *F1 score* represents the harmonic mean of precision and recall, both of which are essential in identifying effective repairs. In general, only patches that achieve an optimal F1 score of 1 are considered suitable for production, as they indicate a complete repair. While partial repairs are theoretically possible, they rarely result in practically useful fixes, due to introducing unwanted behavior or ignoring the underlying cause of the fault. Notably, it is important to acknowledge that likely no metric can fully determine the quality of a patch, as all evaluations are inherently constrained by the incomplete nature of test suites.

5.2 Setup

Reproducibility. This thesis valued the reproducibility of outcomes regarding specific repair configurations. To achieve this objective, several adjustments were implemented. The following aspects played a crucial role in that:

Order of Suggestions. During the implementation, it became evident that different results appeared despite using the same seed for Python's randomization, which should ideally ensure deterministic behavior. A look into the repair process using a debugger revealed that the primary source of this inconsistency lay in how SFLKit [26] returned code lines associated with suspiciousness values (suggestions). Although the suspiciousness values themselves were correctly assigned, the order of suggestions with the same weight was inconsistent. This inconsistency affected FixKit [25], which iterates over code lines and applies mutations by chance—such that the same index of the suggestions received the same mutation. For example, the fifth suggestion might induce a mutation for the specific code line the suggestion identifies. If the fourth and fifth suggestions have the same weight and the order is inconsistent, their position in the list of suggestions might switch. Thus, the mutation is applied for the fifth suggestion which now corresponds to the fourth suggestion from before, affecting a different code line. To address this issue, the suggestions were re-sorted to ensure consistency: first by descending weight and then, in cases of identical weights, by ascending identifier numbers. While this adjustment might lead to a bias toward lower identifier numbers (often corresponding to code lines earlier in a file), it should not affect the current evaluation because it treats the suggestions with the same weight equally.

Order of Files. While the order of suggestions varied across all tested systems, the way how FixKit [25] used the files to create the identifiers, linking specific code lines to suggestions, was environment-dependent. Every experiment that ran on a local machine using Windows Subsystem for Linux (WSL) reported consistent results. However, inconsistencies arose on the Gruenau servers, where the evaluation was conducted, due to differences in which files were read by the statement finder of FixKit. To mitigate this, the statement finder now sorts the files first by lexicographical order.

Tests4Py Evaluation. Finally, the Gruenau server environment also introduced some minor variations in the results, regarding the validation process of the repair. For instance, Tests4Py [24] sometimes evaluated a patch candidate with the mutation `Delete(16)` to have a fitness of 0.09 and other times 0.06. This inconsistency could not be replicated on the local machine and likely indicated an issue within Tests4Py itself or the way the fitness function is created. Although this only happens for a small fraction of candidates, the earlier it deviates, the more different the results are going to be.

Configurations. Table 4 presents an overview of the selected parameters, which includes four variants, two test suite sizes for the baseline, and four test suite sizes for the other variants. This configuration was executed five times, with different seeds used for both the repair process and the generation of the test suite. The test cases for the evaluation set were generated using a fixed seed (0) to ensure that the results from different runs remained as comparable as possible. The baseline employs the specified test suite sizes for both the fault localization and validation portions of the repair process. The fault localization variant only uses the specified test suite sizes for fault localization, otherwise employing the baseline test suite (1, 10) for validation. Conversely, the validation variant always uses (1, 10) for fault localization and incorporates the specified test suites in its validation. The complete variant does not include any test cases from the baseline and uses the specified test suites throughout the entire repair process (both for fault localization and validation).

Table 4: Configuration setup of different parameters with the total number of combinations evaluated.

Variations	Number	Parameters
Repair approaches	1	GENPROG
Variants	4	Baseline, fault localization, validation, complete
Bugs	8	`calculator`, `middle_1`, `middle_2`, `expression` `markup_1`, `markup_2`, `pysnooper_2`, `pysnooper_3`
Iterations	1	10
Baseline tests	2	*Failing* and *passing*: (1, 1), (1, 10)
Additional tests	4	*Failing* and *passing*: (5, 5), (10, 10), (30, 30), (50, 50)
Total runs	5	Seeds (repair): 1714, 3948, 5233, 7906, 9312 Seeds (tests): 959, 2655, 4916, 6114, 8452
Combinations	560	(112 per run, 14 per subject)

As discussed in Section 4.3, the evaluation also examines how altering suggestions could impact the repair effectiveness. The employed modifiers follow the structure of the setup and each one runs with the same configuration as described in Table 4. All configurations were executed with a maximum of 10 iterations and the current generation was extracted whenever the repair process was stopped, indicating whether a patch was found early.

6 Results

This section presents the evaluation results by addressing the following research questions for each subject:

RQ1. How do additional test cases improve different stages of the repair process?

RQ2. How do additional test cases affect the duration of the repair?

RQ3. Do the proposed modifiers described in Subsection 4.3.1 lead to improved repairs for larger test suites compared to the default implementation?

Interpretation of Results. Given the limited sample size of five runs per subject, the results lack statistical significance. However, trends in effectiveness can still be observed. After examining the specific behavior of the repair process, these effects can provide insights into repair effectiveness and overfitting based on the selected subjects, modifiers and test suite sizes. The results for each subject are presented in a table, with the F1 scores being the primary metric, which were calculated by an individual evaluation set containing 50 failing and 50 passing test cases. Whenever a value in parentheses (ranging from 1 to 10) precedes the F1 score, it indicates the generation at which the repair process has stopped. The repair process of FixKit [25] automatically stops upon identifying a valid repair (when the fitness is approximately 1) based on its defined validation. If not stated otherwise, all results primarily present runs with the `TopEqualRankModifier(k=3, threshold=0)` and a general mutation chance `w_mut` of 0.2.

Moreover, for the presentation of plots, either the actual success rate or the overfitting rate is displayed. The actual success rate represents the proportion of runs for a given configuration that resulted in an actual fix. In the case of `middle`, the evaluation test suite is always sufficient to definitively determine whether the fault has been resolved—this was manually verified for all patches that achieved an F1 score of 1. The perceived or tool-reported success rate reflects the proportion of runs that were stopped due to the repair process thinking it found a valid patch, this directly correlates to the evaluation capabilities of the validation test suite. The overfitting rate is calculated by subtracting the perceived success rate from the actual success rate for each configuration, thus presenting the proportion of runs that were mistakenly identified as valid by the repair process but ultimately failed to fix the fault.

From all eight bugs, four groups with two bugs each have emerged that show different behaviors: (6.1) Two bugs (`middle_1` and `middle_2`) were fixed by FixKit, (7.1) for two bugs (`markup_1` and `markup_2`) no appropriate repair was found due to an incorrect implementation of the oracle, (7.2) two bugs (`expression` and `pysnooper_3`) have found a patch that reached a maximum fitness but is incorrect due to an insufficient oracle, and lastly (7.3) two bugs (`pysnooper_2` and `calculator`) are not fixable by typical search-based repair approaches. The following subsections present the results of the subjects that were fixed and assess the research questions, while Section 7 discusses why six bugs were not fixed.

6.1 (RQ1) Repair Improvements

Repair Effectiveness. The repair process for `middle_1` successfully found valid patches and effectively resolved the fault in the program. However, as shown in Table 5, only seed 1714 indicates an increase in the actual success rate with additional tests in fault localization compared to the baseline. For all other seeds, outliers can be observed: In some cases, the baseline performs better (e.g., seed 5233), while in others, the fault localization effectiveness decreases with more tests (e.g., seed 9312). It is likely due to the small impact of improved fault localization being outweighed by the inherent randomness of the repair process. As presented later, `middle_2` exhibits the same effects as `middle_1`, thus to answer *RQ1* partially, one of the key observations is:

Additional test cases did not show a clear improvement in repair effectiveness when applied to fault localization.

Using specific modifiers can reduce this amount of variety caused by the inherent randomness, but does not eliminate it. When comparing these results to those from the original implementation of FixKit [25], using the `DefaultModifier()` and a lower mutation chance (`w_mut=0.06`), the randomness has an even higher impact, as shown in Table 6. Valid repairs are less frequent, and for each seed that tries to enhance fault localization, at least one test case configuration fails to find any useful patch (indicated by the greyed-out 0.00 in the table). The baseline configuration (1, 10) and the validation portion, which employs (1, 10) as the test cases for localizing the fault, outperform other configurations. This difference is likely due to the ratio of test cases, as previously discussed in Section 4.3, which highlights a fundamental flaw of the default implementation when scaling up test suites.

When examining the actual success rates (see Figure 5), it becomes evident that the `DefaultModifier()` works better with the 1:10 ratio for fault localization (see baseline and validation), while the `TopEqualRankModifier()` outperforms it in the complete portion. However, the effectiveness in the fault localization portion appears similar among the two modifiers given the same mutation chance. This is likely the case, because the ineffective validation often led the repair to stop early, thus hiding the actual performance difference for the two modifiers.

Table 5: Results of `middle_1` for `TopEqualRankModifier()` and `w_mut=0.2`.

Variant	Test cases			Seeds		
		1714	3948	5233	7906	9312
Baseline	(1, 1)	0.71	(8) 0.91	0.71	(9) 0.91	0.00
	(1, 10)	0.71	0.71	(6) 1.00	0.81	(8) 0.91
Localization	(5, 5)	(2) 1.00	0.71	0.81	(2) 0.91	0.00
	(10, 10)	(2) 1.00	0.71	(3) 0.91	(2) 0.91	(8) 1.00
	(30, 30)	(2) 1.00	(9) 0.91	0.81	(2) 0.91	(8) 0.91
	(50, 50)	(2) 1.00	(9) 0.91	0.81	(2) 0.91	(5) 0.91
Validation	(5, 5)	0.81	0.71	0.81	0.78	0.00
	(10, 10)	0.81	0.71	0.81	0.91	0.00
	(30, 30)	0.91	0.71	0.81	0.81	0.00
	(50, 50)	0.00	0.71	0.81	0.81	0.00
Complete	(5, 5)	(2) 1.00	0.71	0.91	0.91	0.00
	(10, 10)	(2) 1.00	0.71	(4) 1.00	0.91	(10) 1.00
	(30, 30)	(2) 1.00	(3) 1.00	(10) 1.00	0.91	0.91
	(50, 50)	(6) 1.00	0.81	(4) 1.00	0.91	0.00

Table 6: Results of `middle_1` for `DefaultModifier()` and `w_mut=0.06`.

Variant	Test cases			Seeds		
		1714	3948	5233	7906	9312
Baseline	(1, 1)	0.81	(1) 0.91	(6) 0.91	0.71	0.81
	(1, 10)	(3) 0.91	(6) 1.00	(2) 0.91	(3) 0.91	(7) 0.91
Localization	(5, 5)	(6) 0.91	0.00	0.87	(5) 0.91	0.71
	(10, 10)	(7) 0.91	0.00	0.00	(1) 0.91	0.00
	(30, 30)	0.00	0.00	0.00	(7) 0.91	0.00
	(50, 50)	0.00	(9) 0.91	(8) 0.91	0.00	(6) 1.00
Validation	(5, 5)	0.91	(6) 1.00	0.91	0.91	(7) 0.91
	(10, 10)	0.91	(8) 1.00	(8) 1.00	0.91	0.91
	(30, 30)	0.91	(6) 1.00	0.91	0.81	0.91
	(50, 50)	0.91	(6) 1.00	0.91	0.91	0.91
Complete	(5, 5)	0.91	0.71	(6) 1.00	0.81	0.71
	(10, 10)	0.71	0.71	0.00	0.91	0.81
	(30, 30)	0.81	0.00	0.00	0.00	0.81
	(50, 50)	0.00	(8) 1.00	0.91	0.91	0.00

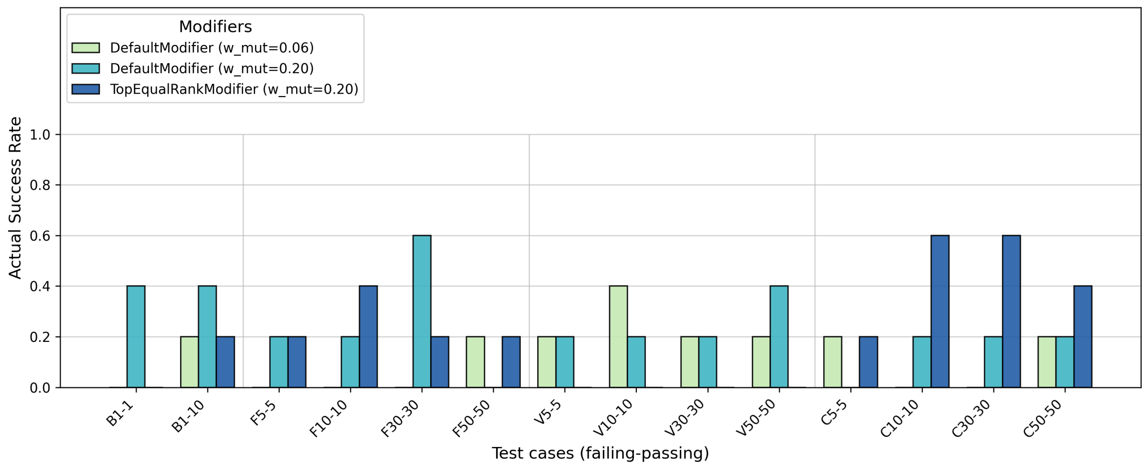


Figure 5: Actual success rate per configuration of `middle_1`.

Overfitting. Table 5 presents a clear reduction in overfitting when comparing the validation and complete portions with the fault localization and baseline portions. With better validation, each time the repair process had stopped (indicated by a displayed generation), the F1 score consistently reached 1, signifying that a valid patch was found. Figure 6 illustrates the overfitting rates for different modifiers and confirms that using additional test cases during validation reduces the occurrence of overfitted patches. The test suite size of 10 failing and 10 passing test cases effectively mitigates overfitting for `middle_1`—using larger test suites only appears to make a marginal difference. Following the observations of several studies [23, 31, 32] and to answer a part of *RQ1*, we confirm that:

Additional test cases can effectively eliminate overfitted patches when used during validation.

Note that the concept of overfitted patches can vary in this context. Since the repair process stops upon identifying an appropriate solution, it may not fully capture the true effectiveness. In principle, the repair could continue running after identifying a correct patch during validation, potentially generating an actual fix by chance later on. The issue lies in the inability of the repair process to accurately assess the quality of a patch without additional test cases. Tools such as Opad [32] incorporate additional test cases to verify whether a certain patch is valid whenever the repair process returns a patch that reaches a maximum fitness score. If this patch fails any of the additional tests, the repair process continues until the patch satisfies the metric Opad uses. This acts like an additional validation layer and does not affect the genetic portion of the repair itself. While this accelerates the repair process, it is not able to refine the internal fitness score. More test cases could lead to a more precise validation, thus enhancing the genetic portion of the repair.

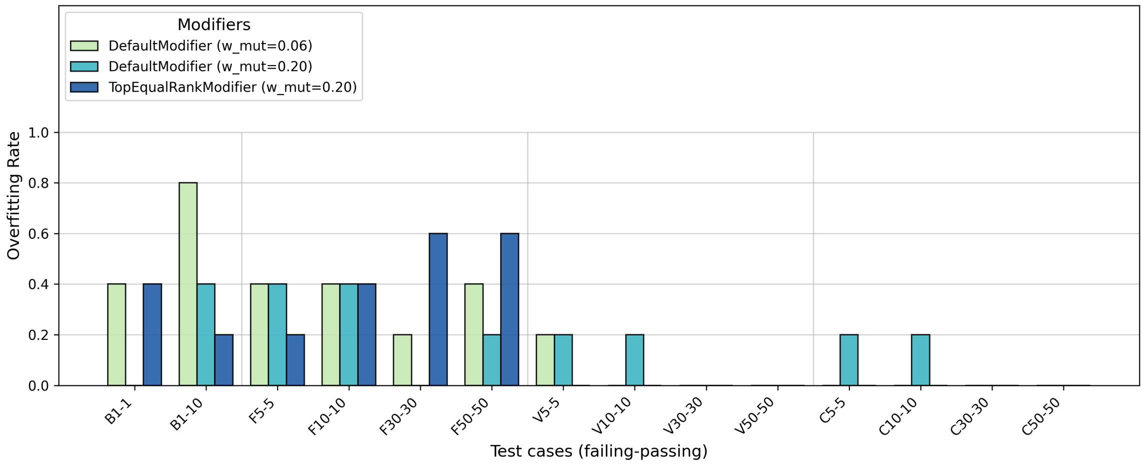


Figure 6: Overfitting rate per configuration of `middle_1`.

In our evaluation, we found that the fitness function does not play a major role in contributing to the effectiveness of the repair. For instance, there are occurrences where the baseline performs better than the validation using larger test suites (e.g. seed 5233). Figure 5 also shows a stable success rate in the validation portion, suggesting that the change in fitness through increased precision is too small to have a significant impact on this subject. A larger program and more iterations are likely required until the genetic refinement is noticeable. A study [13] has also shown that exhaustive repairs, which iterate through a selection of mutations and test each one, can be more effective for many subjects, compared to genetic approaches.

Middle 2. Conversely, `middle_2` could be repaired like `middle_1` and reports even higher success rates, as presented in Table 7 and Figure 7. This is due to `middle_2`

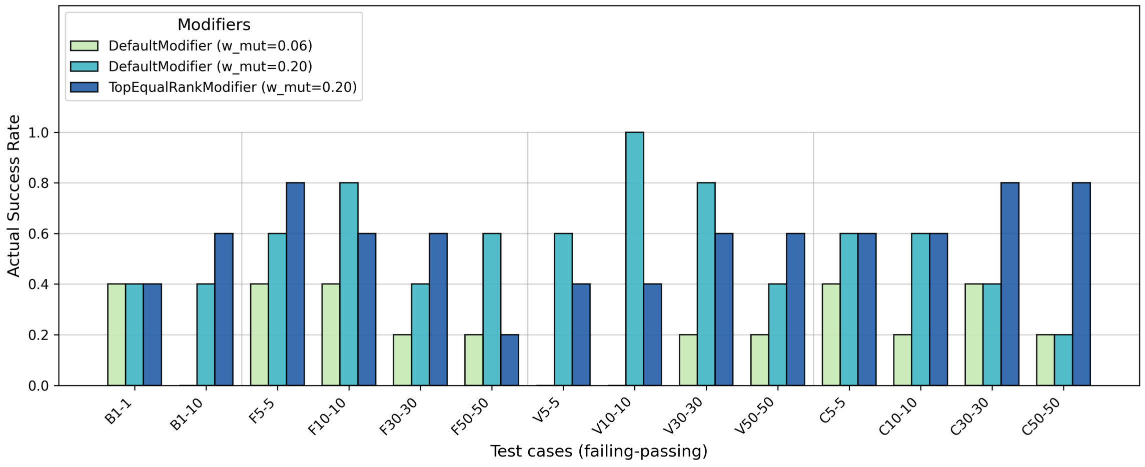


Figure 7: Actual success rate per configuration of `middle_2`.

Table 7: Results of `middle_2` for `TopEqualRankModifier()` and `w_mut=0.2`.

Variant	Test cases	Seeds				
		1714	3948	5233	7906	9312
Baseline	(1, 1)	0.71	(2) 0.91	(1) 1.00	(3) 1.00	(9) 0.91
	(1, 10)	(3) 0.91	(2) 1.00	(1) 1.00	(4) 0.91	(5) 1.00
Localization	(5, 5)	(4) 1.00	(2) 1.00	(1) 1.00	(6) 1.00	0.81
	(10, 10)	(4) 1.00	(2) 1.00	(1) 1.00	0.00	(1) 0.91
	(30, 30)	(4) 1.00	(6) 0.91	(1) 1.00	(7) 1.00	(1) 0.91
	(50, 50)	(6) 0.91	(3) 0.91	(1) 1.00	(7) 0.91	(1) 0.91
Validation	(5, 5)	(6) 1.00	(5) 0.91	(1) 1.00	0.71	(7) 0.91
	(10, 10)	0.91	(2) 1.00	(1) 1.00	0.71	0.71
	(30, 30)	(7) 1.00	(4) 1.00	(1) 1.00	0.81	0.81
	(50, 50)	0.91	0.91	(4) 1.00	(10) 1.00	(10) 1.00
Complete	(5, 5)	(4) 1.00	(2) 1.00	(2) 1.00	0.81	(7) 0.91
	(10, 10)	0.81	(2) 1.00	(1) 1.00	0.81	(2) 1.00
	(30, 30)	(10) 1.00	(5) 1.00	(1) 1.00	0.81	(2) 1.00
	(50, 50)	(7) 1.00	(4) 1.00	(1) 1.00	0.81	(5) 1.00

having a smaller search space, as it does not include a specific `cli.py`-class that is needed for `middle_1`. Nonetheless, `middle_2` experiences the same effects on repair effectiveness as described before, with no clear indication of improving fault localization regarding repair success rates. As for overfitted patches, they are reduced with additional tests during validation as best seen in Figure 8.

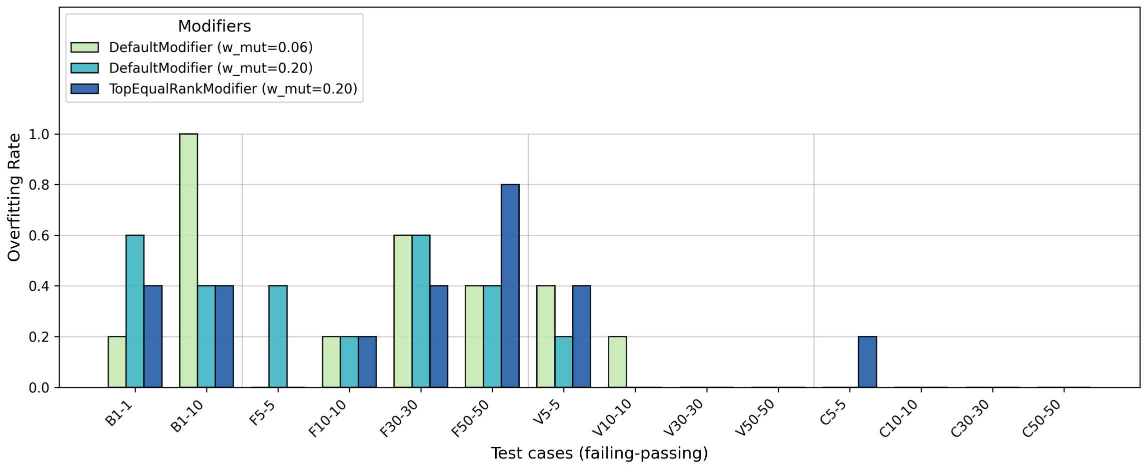


Figure 8: Overfitting rate per configuration of `middle_2`.

One notable difference is that the `DefaultModifier` with `w_mut=0.2` performed prominently better than other modifiers in the validation portion, having the least amount of overfitted test cases. In principle, the choice of modifier should not directly affect the validation process but can influence which mutations are found first. If the correct mutations to fix the fault are applied first, the presence of overfitted patches will decrease naturally. Most overfitted patches happen since the repair looks for the first best possible solution and cannot differentiate the quality between multiple valid patches without additional tests.

Other Subjects. For completeness, Tables 9, 10, 11 and 12 are presented in the appendix of this paper, demonstrating the results for the subjects that did not find a valid fix due to incorrect or insufficient oracles, but still produced high scores in the metric. It should be emphasized that these values do not accurately represent the repairability of those subjects. As for `calculator` and `pysnooper_2`, not a single configuration of test cases led to an F1 score over 0. Section 7 discusses the reasons why these subjects failed.

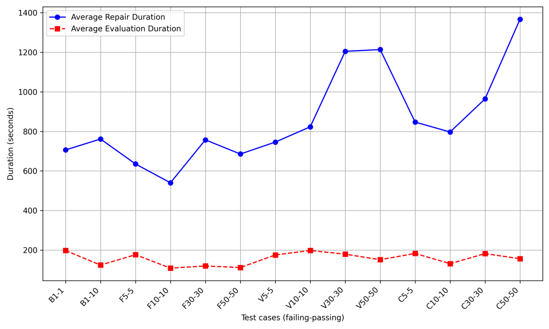
6.2 (RQ2) Repair Time

Figure 9 and Table 8 present the average repair duration, measured from the initialization of FixKit [25] to the repair returning found patches after finalization, as well as the average evaluation duration, which encompasses the final computation of fitness scores and other metrics based on the evaluation set. The graphs depict the mean duration across the five independent runs, all using the `TopEqualRankModifier` with `w_mut=0.2`.

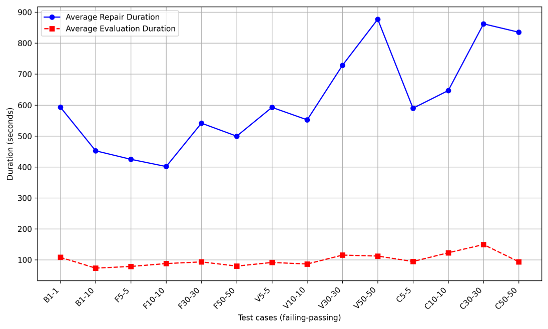
Note. All duration values were measured on the Gruenau servers, which at times experienced high server loads during evaluation. Consequently, direct comparisons across different subjects are not recommended, as the executions may have occurred at different times, thus experiencing varying computational conditions.

The repair duration appears to be strongly influenced by the number of test cases used for validation, which is reasonable given that validation is performed in every iteration, whereas fault localization is executed only once in the beginning. Moreover, most subjects exhibit a stable or slight upward trend in repair duration as the number of test cases for fault localization increases, except for `expression`, where the duration consistently decreases as the test suite size increases in both the baseline and fault localization portions. To answer *RQ2*, the gathered data led to the main observation that:

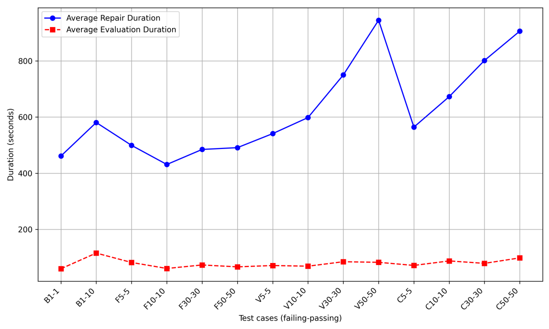
Additional test cases for validation increase
the time required for repairing the fault.



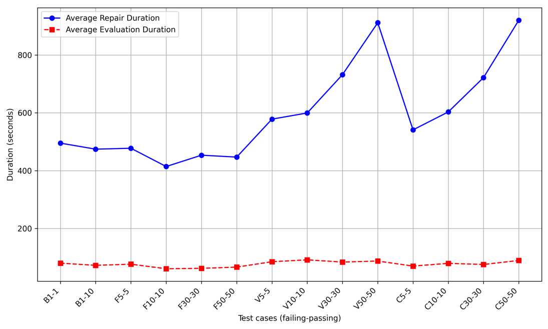
(a) middle_1.



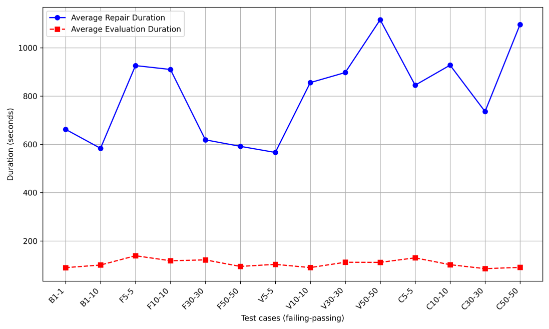
(b) middle_2.



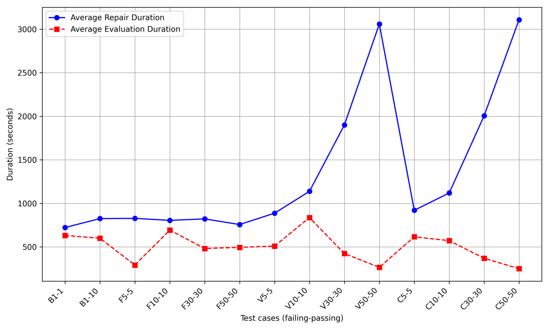
(c) markup_1.



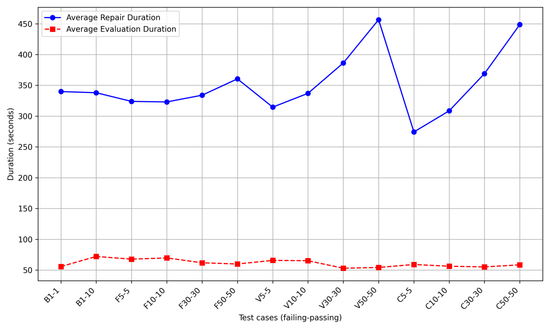
(d) markup_2.



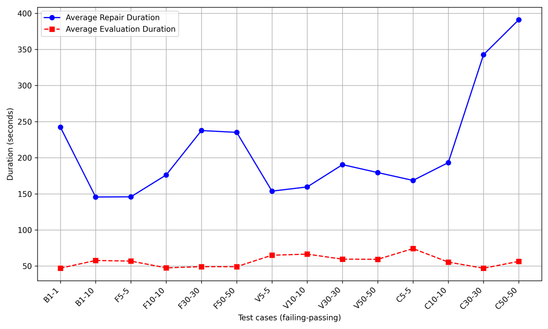
(e) expression.



(f) calculator.



(g) pysnooper_2



(h) pysnooper_3

Figure 9: Average repair and evaluation time per configuration of each subject in seconds.

Table 8: Average repair and evaluation time in seconds.

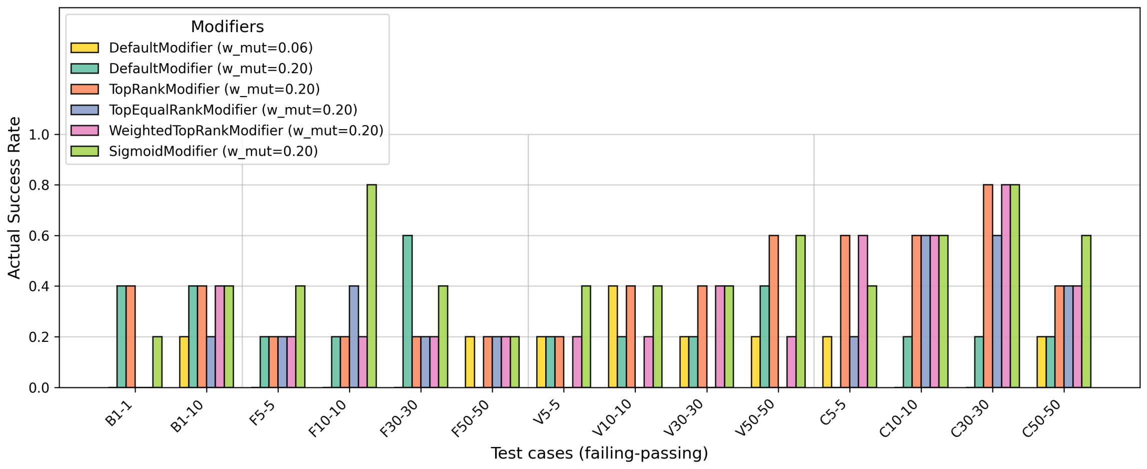
Repair Time														
Subjects	B1-1	B1-10	F5-5	F10-10	F30-30	F50-50	V5-5	V10-10	V30-30	V50-50	C5-5	C10-10	C30-30	C50-50
`middle_1`	706.10	761.58	635.40	540.05	757.24	685.54	746.01	823.33	1204.96	1213.96	847.17	796.95	964.45	1367.26
`middle_2`	593.12	452.26	424.74	401.43	541.52	499.50	592.52	552.23	728.43	877.13	589.75	646.90	862.27	835.17
`markup_1`	461.34	580.72	499.32	431.20	484.77	491.17	541.18	598.37	750.11	945.16	564.26	672.81	801.75	906.22
`markup_2`	495.27	474.39	477.55	414.27	453.48	446.95	578.29	599.57	732.34	911.97	541.13	603.22	722.16	920.67
`expression`	662.40	583.56	926.32	910.29	618.79	591.79	566.51	855.84	897.46	1116.30	845.05	928.11	735.95	1096.16
`calculator`	722.39	824.91	827.75	804.66	821.62	756.98	887.07	1138.08	1900.51	3059.53	920.34	1118.76	2004.85	3108.56
`pysnooper_2`	339.87	337.96	323.90	323.05	333.98	360.49	314.57	337.00	386.34	456.58	274.29	308.59	369.03	448.62
`pysnooper_3`	242.34	145.63	145.88	176.14	237.64	235.12	153.74	159.56	190.41	179.45	168.58	193.24	342.71	391.13
Evaluation Time														
Subjects	B1-1	B1-10	F5-5	F10-10	F30-30	F50-50	V5-5	V10-10	V30-30	V50-50	C5-5	C10-10	C30-30	C50-50
`middle_1`	197.48	123.98	176.43	108.44	119.46	111.38	175.06	197.63	179.13	151.01	182.81	131.02	181.83	156.09
`middle_2`	108.24	73.37	78.80	88.12	93.54	79.98	91.67	86.68	115.52	112.33	94.78	123.04	149.55	93.84
`markup_1`	59.79	115.59	82.13	60.75	73.19	66.48	71.26	68.85	84.77	82.65	71.56	87.63	78.94	98.45
`markup_2`	79.93	72.33	76.36	60.58	61.99	66.14	84.92	91.31	83.68	87.22	69.74	79.22	75.26	89.22
`expression`	89.31	100.15	138.75	117.87	121.47	94.43	102.92	89.60	111.63	111.23	129.92	101.56	85.20	90.33
`calculator`	631.79	599.26	291.66	693.50	481.79	494.75	509.20	836.12	423.28	265.46	615.71	572.02	369.06	250.44
`pysnooper_2`	55.68	72.10	67.68	69.68	61.77	59.86	65.69	65.16	52.95	54.19	59.02	56.20	55.08	58.48
`pysnooper_3`	47.07	57.75	56.86	47.52	49.16	49.17	65.00	66.51	59.54	59.32	74.22	55.46	46.98	56.46

Another noticeable effect is that `calculator` required substantially more time for repair compared to other subjects and was more affected by the number of validation test cases. This trend was also observed in earlier evaluations and runs involving different modifiers. One possible explanation is that `calculator` struggles to identify useful patches, leading to an inefficient, aimless search process. However, `pysnooper_2` follows a similar repair process without ever finding useful mutations and still takes significantly less time. Moreover, `calculator` also presents the highest average evaluation duration among all subjects, suggesting that its validation process may be unoptimized despite it being a relatively small program. On top of that, only `calculator` experiences fluctuations in evaluation duration, whereas all other subjects maintain fairly consistent and stable evaluation times.

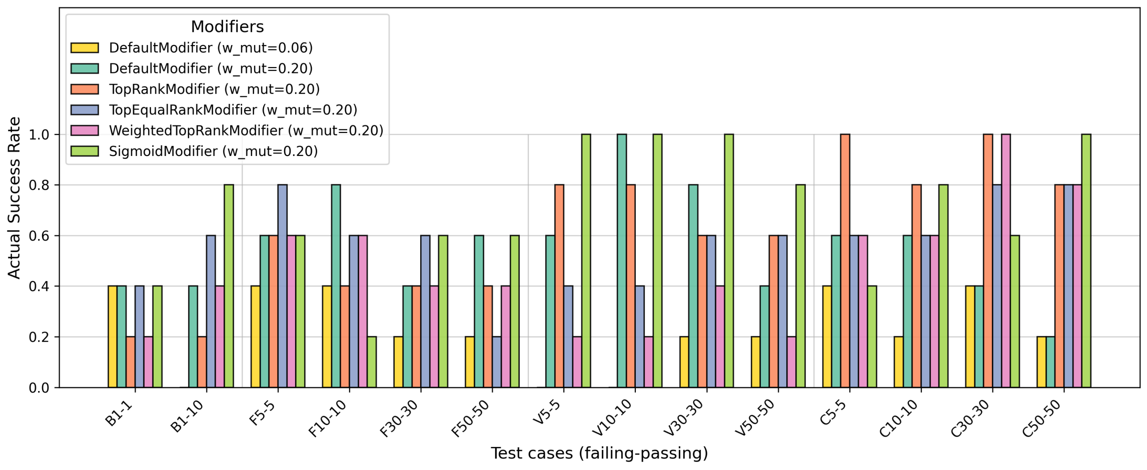
Factoring in the repair effectiveness for `middle_1` and `middle_2`, it appears that 10 failing and 10 passing tests could be the best compromise between repairing the fault effectively and the required time. Increasing the number of validation test cases for these subjects has minimal impact on the repair effectiveness, while leading to notably longer repair durations. However, using more test cases in localizing the fault, especially considering the use of modifiers, can effectively improve repair effectiveness with only slightly slower repairs.

6.3 (RQ3) Effects of Modifiers

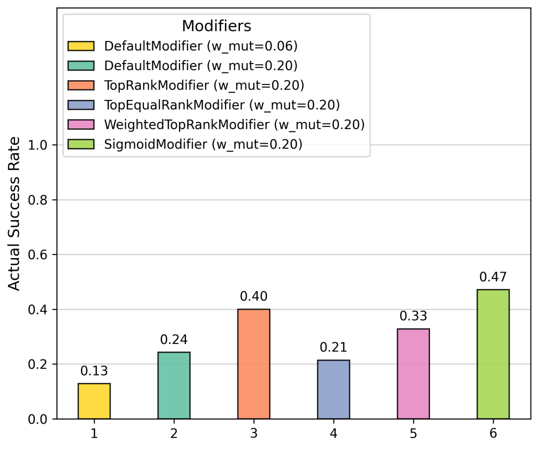
As shown in Subsection 6.1, the `DefaultModifier` is less effective for larger test suites than other modifiers. Figures 10a and 10b display the effectiveness of all modifiers for `middle_1` and `middle_2`, respectively. As for the complete portion with high amounts of test cases, the results of the `DefaultModifier` are falling short compared to the other modifiers. However, as for the baseline and lower test suite sizes, the `DefaultModifier`



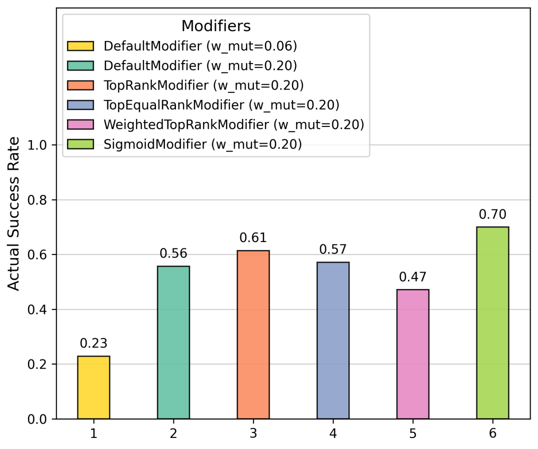
(a) Success rate per configuration of `middle_1`.



(b) Success rate per configuration of `middle_2`.



(c) Total average of `middle_1`.



(d) Total average of `middle_2`.

Figure 10: Comparison of the actual success rates of `middle_1` and `middle_2`.

functions as well as the other modifiers, if not better. When averaging the results of all test suite configurations into a single average success rate, it holds up with other modifiers (see Figures 10c and 10d) given the same mutation chance. Nevertheless, decreasing the mutation chance to 0.06 with the `DefaultModifier` leads to considerably lower success rates.

To accurately see how effective modifying the suggestions truly can be, the subjects should have been evaluated on a range of mutation chances and with close analysis of the locations, where mutations would be applied. Due to the large number of needed parameters, this falls outside the scope of this thesis. We still want to highlight the effectiveness in the form of the shown bar plots, although we did not look into the exact functionality of how each modifier behaved in detail. In conclusion, modifiers were introduced to resolve the degrading performance in larger test suites given the default implementation. Modifying the suggestions accordingly has shown to be an effective way for incorporating a large amount of test cases into the repair process. Despite not finding a clear winner among the proposed modifiers, examining the results for large amounts of test cases led to the observation (which answers *RQ3*) that:

> *The performance of the proposed modifiers scales up more effectively with larger test suites than the default implementation.*

7 Discussion

Despite some bugs achieving maximum F1 scores, the repair did manage to resolve the underlying issue, and for some subjects, it is simply impossible to repair the fault by the methods employed in FixKit [25]. The following subsections discuss the challenges encountered and why many bugs could not be successfully repaired.

7.1 Incorrect Oracle

The results (see Tables 9 and 10) indicate that neither of the bugs in `markup` led to valid fixes, with only partial repairs reaching a maximum F1 score of 0.85. Upon analyzing the generated patches alongside the corresponding test cases, which never passed any patch, it became clear that the oracle of `markup` in Tests4Py [24] was incorrectly implemented. The intended behavior of the subject `markup` is to remove all tags from a simple HTML string. For instance, the input `<a>example</a>` should return `example`. This is achieved by iterating through the characters of the input string while maintaining a state to determine whether the current character is inside a tag. If it is not, the character is appended to the output. Furthermore, the function accounts for cases where a quote appears within a tag. When this occurs, any angle brackets inside the quoted text are ignored (e.g., `<a="<b>">c</a>` becomes `c`).

Markup 1. `markup_1` introduces a bug by incorrectly handling the precedence of boolean operators. Instead of the correct `(c == '"' or c == "'") and tag`, `markup_1` implements it as `c == '"' or c == "'" and tag`, which falsely evaluates the `and` operation first. This results in the function deleting all quotation marks, irrelevant if it is inside or outside a tag, so `<a>"b"</a>` returns `b` instead of `"b"`. Looking through the patches, `markup_1` never found a complete fix, which likely fails due to the limitations of FixKit [25], which is not able to directly correct precedences in booleans without finding a similar code line elsewhere. Nonetheless, `markup_1` reaches a relatively high F1 score of 0.85 by deleting the if-statement of the faulty condition and the subsequent code line. This causes inputs to include quotation marks, thus passing many test cases, but in rare cases, in which there is a quote inside a tag, still fail because the state is not updated properly. For example, `<a="<b>">c</a>` returns `"c`, which is incorrect.

Markup 2. `markup_2`, on the other hand, contains a bug that incorrectly initializes the `tag` state to be true in the beginning, ignoring all characters until a tag is closed. For example, `a<b>c` returns only `c` instead of the correct `ac`. Despite not reaching a maximum F1 score in any repair, `markup_2` still finds a correct fix: It copies the later-occurring code line `tag=False` and places it directly behind `tag=True` at the beginning of the function, thereby resolving the bug. However, this does not reach an F1 score of 1: It appears that the oracle compares the output of `markup` to an expected value. While calculating the expected value, it removes all occurrences of ˆ (circumflex), which does not align with the implementation of `markup` in general. We assume this is simply a fault left in the oracle. Although this renders the results of the evaluation useless, it also highlights one very important aspect: The oracle is just as important as the test suite and can include bugs or be insufficient, as further discussed in 7.2.

7.2 Insufficient Oracle

Besides an oracle being incorrectly implemented, it can also be insufficient for a repair to differentiate valid and invalid patches. When using system test cases, the oracle is responsible for two tasks: (1) verifying that the code runs without an error or an error is appropriately handled, and (2) asserting that the functional behavior of the program is correct. Both tasks may require complex implementations to assess the patches sufficiently, as we will see with `expression` and `pysnooper_3`. The repair reported fixes for both subjects as shown in Table 11 and 12 respectively, by satisfying the oracle without resolving the underlying issue.

Expression. The subject `expression` is a simple parser designed to evaluate mathematical expressions represented as strings based on the operators: *negation*, *addition*, *subtraction*, *multiplication* and *division*. It works similarly to Python's `eval()`-function. The faulty behavior of `expression` is the lack of error handling when dividing by zero. As a result, inputs such as `"5 / 0"` or `"8 / (4 - 4)"` cause the program to fail. The oracle verifies whether the bug was fixed by first analyzing the error output stream. If any error other than a `ValueError` is found, the oracle classifies the input as failing. Next, it compares the output of the program with an expected value, calculated using Python's `eval()`-function. The input passes if both values are the same. The idea behind it is that if a `ValueError` occurs, the oracle assumes that the faulty case was handled correctly, similar to the error handling of Python's `eval()`-function. However, this assumption is misleading. Instead of implementing a correct error handling for the division by zero, the patches that achieved an F1 score of 1 swapped the code line from the parsing of a division to the parsing of a constant, in the form of `Constant(int(token))`. The way the function handles this code line, it always has `/` (division operator) as the token, which causes a `ValueError` when parsing it to an integer. Thus, all cases where a division appears in the expression (not only when dividing through 0) throw a `ValueError`, thus being classified as passing by the oracle. It is unclear, how exactly an oracle should assess how faulty behavior (dividing through zero) should be handled by the program, but it is clear that simply checking for a `ValueError()` is not sufficient to ensure the error was handled correctly. No true repair for `expression` was found by FixKit [25].

Pysnooper 3. Similarly, the repair process for `pysnooper_3` achieved an F1 score of 1 in almost every configuration of test cases. Pysnooper [22] is a small program, which describes itself as a *poor man's debugger*. It uses Python's decorator functionality to mark certain methods, which then log additional information for debugging. Tests4Py [24] implements `pysnooper_3` as a once buggy version of the program—the name of a certain variable was mistakenly written as `output_path` instead of `output`. This causes `pysnooper_3` to throw a `NameError` when trying to use the variable, which is part of the `write()`-function inside `pysnooper_3`. To catch this error, Tests4Py implements Pysnooper as an `ExpectErrAPI`, which is initialized with the following byte string:

```
expected = b"NameError: name 'output_path' is not defined"
```

The API verifies whether this exact sequence of bytes appears in the standard error stream returned by the process which executed the program with the input. If it is part of the error stream, the input is classified as failing, indicating that the exact `NameError` still persists. This also means that as long as no error is thrown, the program passes all inputs, which the repair takes advantage of: For all patches that the repair produced and identified as valid, the code was mutated to remove the execution of the `write()`-function, in which the error occurs, or a function that invokes it. Since this is not the intended fix, it demonstrates that the oracle is insufficient for accurately determining whether the program has been correctly repaired.

This highlights a broader issue when relying on system tests and oracles for validating patches, especially when repairing faults. Due to the repair process altering the behavior of the main program, unforeseen patches can emerge that are unlikely to align with the implemented oracle. This makes creating robust oracles a difficult challenge, even for testing frameworks.

7.3 Impossible to Fix

For the subjects `calculator` and `pysnooper_2`, the repair process failed to generate any valid patches—whether complete or partial—that fixed the fault, as indicated by a consistent F1 score of 0. This failure is related to the *competent programmer assumption* [8], a common premise in many search-based repair techniques. This assumption states that the correct mutation can be inferred from the surrounding code, based on the idea that most faults result from small deviations of the intended implementation that are easy to overlook. This assumption greatly reduces the search space of possible mutations, while also proving to be an effective solution for some subjects. However, this property also limits the capabilities of the repair process, as we see in the following subjects:

Calculator. The fault in `calculator` is impossible to be repaired by FixKit [25] due to `calculator` not containing the required code lines to repair itself. In detail, the bug is derived from a custom implementation of the `sqrt()`-function with the Newton-Raphson method. When parsing negative input values, the function will raise a `ZeroDivisionError`, from which the corresponding oracle classifies the input as failing. Similar to the oracle of `expression`, it specifically searches for error-handling that is compromised of either a `ValueError` or `NameError` for inputs to pass. This would align with the standard implementation of the `sqrt()`-function from Python's math module to pass all test cases. However, the oracle itself might be incomplete for correctly verifying patches. For example, it does not consider raising an *AssertionError* as valid error handling, even though that would also be a reasonable approach.

Nevertheless, due to competent programmer assumption, `calculator` is not able to raise any of the errors and thus cannot be fixed by most search-based repair techniques.

Pysnooper 2. `pysnooper_2` suffers from a fault caused by the absence of handling for a specific input parameter, named `custom_repr`, which leads to an error. Resolving this issue would require modifying the function signatures across multiple methods to incorporate this parameter. However, since no existing code references `custom_repr`, the repair process is unable to infer the necessary changes, making it infeasible for the repair to fix `pysnooper_2`. In addition to this limitation, `pysnooper_2` presents two further challenges:

1. Successfully repairing the fault requires applying multiple instances of the same type of mutation across different locations in the code. However, since FixKit [25] selects mutation types randomly, it becomes a combinatorial problem. The genetic approach makes it very unlikely to select all the right mutations for the same patch.

2. `pysnooper_2's` fault localization provides minimal information, assigning weights of only 0 or 1 to code lines. This occurs because the error consistently arises when the missing parameter is part of the input, which is caught at the beginning

of the program. Consequently, fault localization fails to accurately pinpoint the root cause of the issue, further reducing the likelihood of a successful repair.

All these factors demonstrate important challenges to consider when trying to resolve defects using search-based repairs.

Solvability. The previous subsections highlighted the limitations of incorrect and insufficient oracles across multiple subjects. However, even if all oracles were correctly implemented, we suspect that only `middle_1`, `middle_2`, and `markup_2` are solvable among the eight total subjects.

8 Threats to Validity

The evaluation has the following threats to validity:

Internal Validity. The threats to internal validity relate to the correctness of the evaluation pipeline and other tools used for the evaluation, such as FixKit [25] and Tests4Py [24], which are still experimental and lack thorough testing for validity. Due to the high amount of parameters and unpredictable results, it would be cumbersome to verify the correctness in detail. During the implementation, verbose logging and debuggers were used to assert that mutations were correctly applied and that fault locations were modified according to the specified modifiers. However, we are concerned about the correctness of Tests4Py, since the validation results led to irregular fitness scores on the server environment, as stated in Subsection 5.2 about reproducibility. The same applies to the correctness of the oracles, implemented in Tests4Py. As discussed in Section 7, some oracles appear to behave incorrectly regarding the intended behavior of resolving the fault. Although the oracle and produced patches of all subjects were analyzed, we cannot guarantee the validity beyond what is stated due to the sheer amount of configurations and different approaches that went into this evaluation.

External Validity. This thesis worked under the initial impression to evaluate more real-world defects, which would have been provided by Tests4Py [24]. During early testing, it was examined that many of the possible subjects were not effective in either generating inputs or being repaired automatically. Since only the toy subject `middle` produced valid results, this evaluation is not generalizable for real-world programs—especially in the use of the proposed modifiers and mutation chances, as they likely behave differently when scaled to larger programs. Additionally, other configurations such as varying the mutation chance or testing different ratios of failing to passing test cases would be necessary to evaluate the effects further and confirm proposed claims.

9 Conclusion and Future Work

This thesis evaluates the impact of incorporating additional test cases in the automated program repair process. To gain detailed insights, the effects of test cases on fault localization and validation were separately examined. The results indicate that additional test cases for validation effectively reduced the occurrence of overfitted patches—patches that only satisfy the test suite without correcting the underlying fault in the program. However, no significant improvement regarding the refinement of the genetic process was observed.

Furthermore, increasing the number of test cases in fault localization revealed that the ratio of failing to passing test cases plays a crucial role in computing the suspiciousness score. When using the OCHIAI metric [1], it favors more passing tests, leading to better repairs. To mitigate these effects, modifiers were introduced to FixKit [25] that alter the way weights for code locations are incorporated into the repair process. Although the modifiers do not increase the average true success rate over all configurations, they can effectively utilize larger test suites, which underperform using the default implementation with no modifications to the suggestions.

Additionally, the results indicate that incorporating more test cases for validation significantly increased repair time, whereas additional test cases for fault localization had only a marginal impact on the duration. In conclusion, this thesis confirms previous findings [31, 32] regarding the reduction of overfitted patches and identifies the ineffectiveness of fault localization to be related to the ratio of failing to passing tests in larger test suites.

Besides general improvements to the correctness and validity of the used approaches, particularly regarding correct oracles and reproducible outcomes, we find the following to be interesting future work:

Different Ratios of Test Cases. The evaluation only considered a very limited amount of different ratios of test cases, with most configurations using a 1:1 ratio and the baseline demonstrating the effects of a 1:10 ratio. Future evaluations could examine the effects of incorporating various ratios of test cases to present which ratio would be most effective in finding repairs.

Alternative Metrics. Instead of modifying fault localization suggestions post hoc, alternative metrics other than the OCHIAI [1] metric might infer interesting insights. They may exhibit reduced dependency on the ratio of test cases or even benefit from additional failing tests. Other common metrics to evaluate are TARANTULA [10] or JACCARD [1].

Mutation Chance. The general mutation chance `w_mut` has been shown to notably impact the effectiveness of how many runs produce a valid patch. Evaluating different mutation probabilities could provide further insights into the effectiveness of specific metrics or modifiers.

Other Types of Program Repair. Although other types of program repair may utilize test cases in distinct ways, many key observations of this thesis can still be applied. Evaluating *semantic-based*, *pattern-based* or *learning-based repairs* with the addition of larger test suites and various configurations may provide insights that can generally be applied to the field of automated program repair.

10 Availability of Data

All experiments and results for this thesis are available publicly. In Detail, the results of the evaluation can be found in the form of CSV-files, containing the metrics for all configurations, and detailed reports with the produced patches for every single run. The current version can be found here:

`https://github.com/MarwinLinke/additional_tests_fixkit`

References

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[3] B. Berabi, J. He, V. Raychev, and M. Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR, 2021.

[4] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[5] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 65–74. IEEE, 2010.

[6] M. Eberlein. Debugging benchmark, 2021.

[7] M. Eberlein, M. Smytzek, D. Steinhöfel, L. Grunske, and A. Zeller. Semantic debugging. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 438–449, 2023.

[8] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 189–200. IEEE, 2014.

[9] R. Gupta, S. Pal, A. Kanade, and S. Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31, 2017.

[10] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[11] A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1228–1239, 2020.

[12] 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)*, pages 802–811. IEEE, 2013.

[13] X. Kong, L. Zhang, W. E. Wong, and B. Li. Experience report: How do techniques, programs, and tests impact automated program repair? In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 194–204. IEEE, 2015.

[14] Y. Lei, C. Sun, X. Mao, and Z. Su. How test suites impact fault localisation starting from the size. *IET software*, 12(3):190–205, 2018.

[15] Y. Li, S. Wang, and T. N. Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th international conference on software engineering*, pages 511–523, 2022.

[16] M. Martinez and M. Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *Search-Based Software Engineering: 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings 10*, pages 65–86. Springer, 2018.

[17] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[18] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[19] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 191–201, 2013.

[20] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[21] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[22] R. Rachum, A. Hall, I. Yanokura, et al. Pysnooper: Never use print for debugging again, jun 2019.

[23] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543, 2015.

[24] M. Smytzek, M. Eberlein, B. Serce, L. Grunske, and A. Zeller. Tests4py: A benchmark for system testing. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 557–561, 2024.

[25] M. Smytzek, M. Eberlein, K. Werk, L. Grunske, and A. Zeller. Fixkit: A program repair collection for python. 2024.

[26] M. Smytzek and A. Zeller. Sflkit: A workbench for statistical fault localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1701–1705, 2022.

[27] D. Steinhöfel and A. Zeller. Input invariants. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 583–594, 2022.

[28] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366. IEEE, 2013.

[29] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374. IEEE, 2009.

[30] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, et al. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1556–1560, 2020.

[31] Q. Xin and S. P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, pages 226–236, 2017.

[32] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 831–841, 2017.

[33] B. Yu, H. Qi, Q. Guo, F. Juefei-Xu, X. Xie, L. Ma, and J. Zhao. Deeprepair: Style-guided repairing for deep neural networks in the real-world operational environment. *IEEE Transactions on Reliability*, 71(4):1401–1416, 2021.

[34] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. Test case generation for program repair: A study of feasibility and effectiveness. corr, abs/1703.00198, 2017. *arXiv preprint arXiv:1703.00198*, 2017.

[35] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. The fuzzing book, 2021.

Appendix

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 20. Februar 2025 Marwin Linke

Table 9: Results of `markup_1`.

Variant	Test cases	Seeds				
		1714	3948	5233	7906	9312
Baseline	(1, 1)	0.19	(6) 0.79	(9) 0.85	(4) 0.85	(2) 0.85
	(1, 10)	0.51	0.51	0.85	0.51	0.41
Localization	(5, 5)	(1) 0.85	(8) 0.85	(8) 0.85	0.85	0.62
	(10, 10)	(9) 0.85	(3) 0.85	(3) 0.85	0.62	(3) 0.63
	(30, 30)	(2) 0.85	(3) 0.85	(6) 0.85	(7) 0.85	(4) 0.63
	(50, 50)	0.51	(8) 0.85	(2) 0.85	0.85	(5) 0.63
Validation	(5, 5)	0.35	0.28	0.17	0.79	0.85
	(10, 10)	0.51	0.40	0.40	0.51	0.79
	(30, 30)	0.51	0.51	0.85	0.51	0.44
	(50, 50)	0.51	0.51	0.51	0.41	0.85
Complete	(5, 5)	0.85	0.30	0.17	0.85	0.19
	(10, 10)	0.46	0.79	0.79	0.51	0.19
	(30, 30)	0.51	0.62	0.48	0.85	0.85
	(50, 50)	0.51	0.51	0.51	0.51	0.62

Table 10: Results of `markup_2`.

Variant	Test cases	Seeds				
		1714	3948	5233	7906	9312
Baseline	(1, 1)	0.80	0.35	(2) 0.80	(4) 0.80	(2) 0.80
	(1, 10)	(5) 0.80	0.80	(2) 0.80	(5) 0.80	(2) 0.80
Localization	(5, 5)	(1) 0.80	(2) 0.80	0.80	(2) 0.80	(4) 0.80
	(10, 10)	(1) 0.80	(2) 0.80	(1) 0.80	(1) 0.80	(4) 0.80
	(30, 30)	(1) 0.80	(2) 0.80	(1) 0.80	(7) 0.80	(8) 0.80
	(50, 50)	(2) 0.80	(3) 0.80	(3) 0.80	(5) 0.80	(5) 0.80
Validation	(5, 5)	0.46	0.80	0.44	0.80	0.80
	(10, 10)	0.46	0.43	0.80	0.43	0.80
	(30, 30)	0.80	0.80	0.46	0.85	0.80
	(50, 50)	0.46	0.80	0.80	0.80	0.80
Complete	(5, 5)	0.43	0.80	0.43	0.45	0.80
	(10, 10)	0.43	0.80	0.85	0.80	0.85
	(30, 30)	0.85	0.80	0.80	0.43	0.80
	(50, 50)	0.80	0.80	0.46	0.46	0.80

Table 11: Results of `expression`.

Variant	Test cases	Seeds 1714	3948	5233	7906	9312
Baseline	(1, 1)	0.00	0.00	0.00	0.00	0.97
	(1, 10)	0.00	0.00	0.00	(9) 0.97	0.00
Localization	(5, 5)	0.00	0.00	0.00	(6) 1.00	0.00
	(10, 10)	(10) 1.00	0.00	0.00	0.00	0.00
	(30, 30)	0.00	0.00	0.00	(7) 1.00	0.00
	(50, 50)	(9) 1.00	0.00	0.00	(7) 1.00	(10) 1.00
Validation	(5, 5)	(2) 1.00	0.97	0.00	(7) 1.00	(4) 0.97
	(10, 10)	(7) 1.00	0.81	0.97	0.81	0.81
	(30, 30)	(8) 1.00	0.00	0.00	(10) 1.00	(9) 0.97
	(50, 50)	(4) 1.00	0.97	0.00	(10) 1.00	(10) 1.00
Complete	(5, 5)	0.00	0.00	0.00	(8) 1.00	(8) 0.89
	(10, 10)	(3) 1.00	0.00	0.00	0.00	0.00
	(30, 30)	0.00	0.00	0.00	(7) 1.00	(7) 1.00
	(50, 50)	0.00	0.00	0.00	0.00	(10) 1.00

Table 12: Results of `pysnooper_3`.

Variant	Test cases	Seeds 1714	3948	5233	7906	9312
Baseline	(1, 1)	0.00	(6) 1.00	0.00	0.00	0.00
	(1, 10)	(1) 1.00	(2) 1.00	(1) 1.00	(1) 1.00	(1) 1.00
Localization	(5, 5)	(1) 1.00	(1) 1.00	(1) 1.00	(1) 1.00	(1) 1.00
	(10, 10)	(1) 1.00	(5) 1.00	(1) 1.00	(1) 1.00	(1) 1.00
	(30, 30)	(4) 1.00	(5) 1.00	0.00	(1) 1.00	0.00
	(50, 50)	(4) 1.00	(5) 1.00	0.00	(1) 1.00	0.00
Validation	(5, 5)	(1) 1.00	(2) 1.00	(1) 1.00	(1) 1.00	(1) 1.00
	(10, 10)	(1) 1.00	(2) 1.00	(1) 1.00	(1) 1.00	(1) 1.00
	(30, 30)	(1) 1.00	(2) 1.00	(1) 1.00	(1) 1.00	(1) 1.00
	(50, 50)	(1) 1.00	(2) 1.00	(1) 1.00	(1) 1.00	(1) 1.00
Complete	(5, 5)	(1) 1.00	(2) 1.00	(1) 1.00	(1) 1.00	(1) 1.00
	(10, 10)	(1) 1.00	(5) 1.00	(1) 1.00	(1) 1.00	(1) 1.00
	(30, 30)	(4) 1.00	(5) 1.00	0.00	(1) 1.00	0.00
	(50, 50)	(4) 1.00	(5) 1.00	0.00	(1) 1.00	0.00