Improving Patch Correctness Analysis via Random Testing and Large Language Models

Facundo Molina
IMDEA Software Institute
Madrid, Spain
facundo.molina@imdea.org

Juan Manuel Copia
IMDEA Software Institute and
Universidad Politécnica de Madrid
Madrid, Spain
juanmanuel.copia@imdea.org

Alessandra Gorla
IMDEA Software Institute
Madrid, Spain
alessandra.gorla@imdea.org

Abstract—**Patch correctness assessment represents a crucial step in the patch validation process, with the potential to enhance the practical adoption of automated program repair (APR) techniques and substantially reduce validation costs. While some automated techniques have been proposed for assessing patch correctness, they primarily focus on either ranking patches based on their likelihood of being correct or classifying them as correct or incorrect without offering any further explanatory information. In this paper, we introduce FIXCHECK, a novel approach that combines random testing and large language models to automatically generate fault-revealing tests for potentially incorrect patches. To achieve this, FIXCHECK employs a two-fold process: Firstly, a random testing procedure generates a comprehensive set of test cases. Secondly, a large language model is utilized to derive meaningful assertions for each test case. Additionally, FIXCHECK incorporates a selection and prioritization mechanism, which evaluates the generated tests executed on the patched program and discards or ranks them based on their likelihood of revealing faults in the patch.**

To assess the effectiveness of our approach, we conducted evaluations on a benchmark comprising 160 patches, encompassing both patches created by developers and patches generated by APR tools. The results demonstrate that FIXCHECK effectively generates fault-revealing tests for 62% of incorrect patches written by developers, with a high level of confidence. Furthermore, it complements existing patch correctness assessment techniques by providing fault-revealing tests for up to 50% of the incorrect patches identified by state-of-the-art techniques.

I. INTRODUCTION

Generating patches that actually *fix* software defects is a crucial task in the maintenance of software systems. Typically, software defects are reported via specifications, commonly in the form of test cases, which unveil undesirable behaviors in the software. In response to these defects, developers create patches that must undergo validation before being committed to the codebase, ensuring that the provided test no longer exposes the defect. However, despite such efforts, these patches may still fail to effectively address the underlying bug or, worse, introduce new bugs, resulting in what is known as bad fixes [16] or incorrect patches. Detecting these erroneous patches can significantly impact the time and effort dedicated to bug fixes by developers and the overall maintenance of software systems.

Automated program repair (APR) has been a very active research area in the last decade [4], [5], [7], [8], [10], [12], [13], [15], [17], [18], [22], [24], [27], [29]–[31], [35]–[37], [47]. APR techniques equip software practitioners with tools capable of automatically generating patches for buggy programs. Typically, these methods operate in a test-based environment, where a faulty program and a test suite containing at least one fault-revealing test are provided. The APR techniques then aim to produce a patched program that successfully passes the entire test suite. When a patch successfully passes the whole test suite, it is deemed *plausible*, and if it indeed resolves the bug, it is considered *correct*. However, since test suites generally offer only a partial specification of the desired software behavior [40], plausible patches generated by APR techniques are prone to overfitting, leading to the creation of numerous *incorrect* patches that fail to address the bug [14], [41]. This issue has been a significant obstacle to the adoption of APR techniques by software practitioners [43]. Therefore, the automated detection of incorrect patches assumes even greater importance.

To tackle this challenge, and with the primary objective of promoting the widespread adoption of APR tools, a range of techniques for patch correctness assessment have been proposed [11], [25], [32], [43], [48], [49], [52]. One category of these techniques consists of *dynamic* approaches that operate by comparing the run-time behavior of the patched program with its buggy counterpart. Among the state-of-the-art dynamic techniques, PATCH-SIM [49] stands out. PATCH-SIM is a similarity-based technique that generates new test cases to measure the behavioral differences between the buggy and patched versions. It then uses the enhanced test suite to assess patch correctness considering that a correct patch may behave similarly on passing tests while differently on failing tests compared with the buggy program. Other techniques for patch correctness analysis are instead based on *static* or *hybrid* approaches. Static approaches address the problem by attempting to *predict* whether a patch is correct or not, based on some features that can be computed with static analysis. As an example, BATS [43] represents a static approach that operates by evaluating the similarity of generated patches with previously verified correct patches. These correct patches correspond to failing test cases that bear resemblance to the failing tests associated with the specific bug under consideration.

On the other hand, hybrid techniques combine static and dynamic information to predict the correctness of a patch. A recent hybrid technique, Shibboleth [11], assesses the correctness of a patch through a combination of three metrics: first, a syntactic similarity between the buggy and patched programs; second, a semantic similarity between the execution traces of both programs; and third, the code coverage achieved by the originally passing tests. Shibboleth subsequently leverages these metrics to rank patches or classify them as correct/incorrect using machine learning algorithms. Despite the significant progress made by these techniques, notably by Shibboleth [11], they still exhibit limitations in the output they provide to users. While assessing the correctness of a patch, these techniques typically yield a binary result (correct/incorrect) without providing any additional information regarding the rationale behind such decision.

Although certain dynamic techniques [49], [52] employ test generation to discover new tests for the patch under analysis, they are still not able to produce tests evidencing the incorrectness of the patch or they only do so in a very limited manner. For instance, PATCH-SIM [49] generates new tests using Randoop [39], uses them only to measure the behavioral differences between the buggy and patched versions, and determines patch correctness from such differences. Thus, when a patch is flagged as incorrect, users may find themselves compelled to invest considerable time and effort in understanding the reasons behind the inaccuracy and devising a concrete scenario (i.e., a test case) that demonstrates the incorrectness of the patch. Opad [52], another dynamic technique based on test generation, uses instead fuzzing testing to generate new test cases on the buggy program, and then relies on two predetermined oracles to detect patches that produce crash or memory-safety problems. However, these oracles are not able to detect other types of incorrect patches. This scenario poses challenges to the effective utilization of these techniques.

To address such limitations, in this paper we propose FIXCHECK, a novel approach for improving the output of patch correctness analyses. FIXCHECK combines static analysis, random testing and large language models (LLMs) to automatically generate tests highlighting and explaining the incorrectness of a patch. The fundamental hypothesis underlying FIXCHECK is that the scenario that reveals the incorrectness of the patch is similar to the original fault-revealing test case. Thus, instead of using standard test generation tools, FIXCHECK employs a static analysis process on the initial fault-revealing test that generates new similar tests by means of *transformations*. These transformations preserve most of the structure of the initial test, and produce a slight modification in the test input. The obtained tests can then be executed against the patched program. If a new test does not trigger a failure, it may be due to the absence of an adequate oracle. To address this issue, rather than using predetermined oracles, FIXCHECK relies on LLMs to generate meaningful assertions to complement the generated tests. The use of LLMs in this task is motivated by their success in a variety of tasks, including program repair [8], [18], [47]. Finally, FIXCHECK collects the traces of all the failing tests (either without or with assertions) and computes a similarity score with respect to the trace of the initial fault-revealing test. Failures with a similarity score above a certain threshold are reported by FIXCHECK as evidence of the incorrectness of the patch under analysis.

We evaluate FIXCHECK on a benchmark composed of 160 patches, 30 of which are correct and 130 are incorrect. These patches come from two sources. First, we use BF4J, a dataset of 40 incorrect patches that we collected from open source projects, for which test cases evidencing their incorrectness are available for 21 of them. This ground truth of incorrect patches and tests is used to accurately assess the performance of FIXCHECK in generating tests evidencing the incorrectness of a patch. Second, we use the dataset of 139 patches used in the evaluation of PATCH-SIM [49]. These are patches for bugs in Defects4J [20], generated with existing APR tools. We use this dataset to analyze how FIXCHECK can be used to improve the output of existing patch correctness analysis techniques. Overall, our results show that FIXCHECK is able to generate tests evidencing the incorrectness of a patch for 62% of incorrect patches written by developers. Moreover, we show that FIXCHECK is able to generate fault-revealing tests for up to 50% of incorrect patches detected by patch correctness assessment techniques, thus improving the output of these analyses. These results provide initial and strong evidence that FIXCHECK is effective, and can provide a valuable output to users of patch correctness analysis techniques.

To sum up, the contributions of this paper are the following:

- FIXCHECK, a novel approach for improving patch correctness analyses that combines static analysis, random testing and large language models.
- BF4J (Bad Fixes for Java), a dataset composed of 40 incorrect patches written by developers during the maintenance of open source projects.
- A detailed evaluation of the performance of FIXCHECK to generate tests evidencing the incorrectness of a patch, and how it can be used in combination with existing patch correctness assessment techniques.

II. MOTIVATING EXAMPLE

This section presents a motivating example with the purpose of illustrating how FIXCHECK can obtain tests evidencing the incorrectness of a patch.

Figure 1 illustrates tests that are part of two subsequent commits made by a developer while fixing a bug in the `jackson-databind` component of the Jackson JSON library for Java. The bug, reported in the issue 118[1], is related to a serialization issue that arises when an external class is being serialized. Figure 1(a) shows the initial fault-revealing test that allows to reproduce the bug. The test first initializes an `ObjectMapper`, which provides methods to serialize/deserialize objects to/from the JSON format. To reproduce the bug, the test attempts to serialize an object of class `ExternalTypeWithNonPOJO`, which has a unique field `value` of class `Object`, in the test

[1]https://github.com/FasterXML/jackson-databind/issues/118

```java
public void testWithScalar118() {
  ObjectMapper mapper = new ObjectMapper();
  ExternalTypeWithNonPOJO input = new
     ExternalTypeWithNonPOJO(new Date(123L));
  String json = mapper.writeValueAsString(input);
  assertNotNull(json);
  // and back just to be sure:
  ExternalTypeWithNonPOJO result = mapper.readValue(json
     , ExternalTypeWithNonPOJO.class);
  assertNotNull(result.value);
  assertTrue(result.value instanceof java.util.Date);
}
```

(a) Initial fault-revealing test.

```java
public void testWithNaturalScalar118() {
  ObjectMapper mapper = new ObjectMapper();
  ExternalTypeWithNonPOJO input = new
     ExternalTypeWithNonPOJO(Integer.valueOf(13));
  String json = mapper.writeValueAsString(input);
  assertNotNull(json);
  // and back just to be sure:
  ExternalTypeWithNonPOJO result = mapper.readValue(json
     , ExternalTypeWithNonPOJO.class);
  assertNotNull(result.value);
  assertTrue(result.value instanceof Integer);
}
```

(b) Subsequent developer-written test.

```java
public void test0() {
  ObjectMapper mapper = new ObjectMapper();
  ExternalTypeWithNonPOJO input = new
     ExternalTypeWithNonPOJO(new Integer(54));
  String json = MAPPER.writeValueAsString(input);
  // and back just to be sure:
  ExternalTypeWithNonPOJO result = MAPPER.readValue(json
     , ExternalTypeWithNonPOJO.class);
  assertNotNull(result.value);
  assertTrue(result.value instanceof Integer);
  assertEquals(Integer.valueOf(54), result.value);
}
```

(c) Test produced by FIXCHECK.

Fig. 1: A bug revealing test in the jackson-databind library and two tests revealing a defect in the corresponding patch: one produced by the developer and the other by FIXCHECK.

being instantiated with a `Date` object. The execution of the test fails with a `JsonGenerationException` that is thrown during the invocation of the `writeValueAsString` method. From the discussion in the issue, one can see that the problem is related to the use of a `Date` object in the `ExternalTypeWithNonPOJO` class. The initial fault-revealing test and the first patch attempting to fix the bug are part of the commit `69a1eae`.

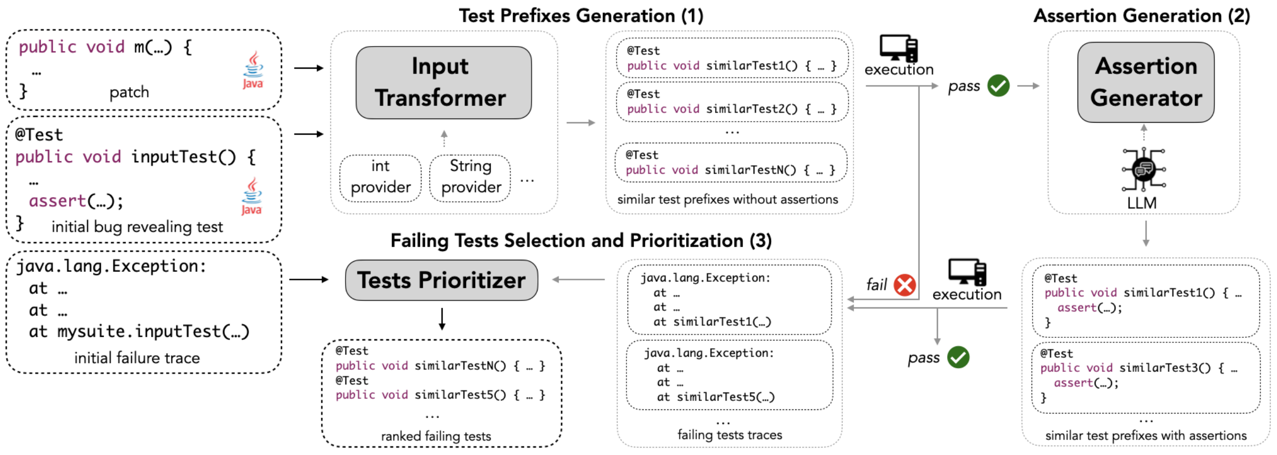
Subsequently, the issue was reopened because the first patch was actually incorrect. Although the patch fixed the problem related to the `Date` object, it did not consider the case in which the object of class `ExternalTypeWithNonPOJO` was instantiated with other types. Figure 1(b) shows a test that reveals the defect in the patched version. It is easy to see the high level of similarity between the initial fault-revealing test and the subsequent test evidencing the defect in the patch. In fact, the only difference between these two tests are: the use of an `Integer` object instead of a `Date` object when instantiating the `ExternalTypeWithNonPOJO` object, and the last assertion that checks that the deserialized object has an `Integer` value instead of a `Date` value. This test was provided by the developer in the commit `a795fa2`, which also provides the final correct patch. This situation is consistent with our intuition that the initial tests that reveal a bug and the tests that allow to exhibit a defect on a patch are often very similar. In fact, in BF4J, the dataset we collected, we found that for ∼70% of the incorrect patches, the test exposing the problem in the patch only differs from the initial fault-revealing test in the test input and in the corresponding assertions. Despite that the incorrectness of the first patch in our example could have been detected by some existing patch correctness assessment technique, such as PATCH-SIM [49] or Shibboleth [11], the test case revealing the defect in the patch would still have to be manually provided.

The main goal of FIXCHECK is to assist developers in producing fault-revealing tests for incorrect patches. To achieve this, it employs a combination of static analysis, random testing and large language models. FIXCHECK works as follows. Initially, it uses a lightweight static analysis to generate a set of tests that differ from the initial fault-revealing test in the test inputs, which are selected through a random process. Then, it leverages LLMs to generate meaningful assertions for each test, which are then executed against the patch. Finally, FIXCHECK compares the failure traces of the failing tests with the failure trace of the initial fault-revealing test, in order to rank them according to the failure similarity.

For our example, FIXCHECK generated the test in Figure 1(c). Similarly to the developer written test in Figure 1(b), the one produced by FIXCHECK instantiates the `ExternalTypeWithNonPOJO` object with an `Integer` object, which allows to reproduce the failure in the patched version. The test also includes the corresponding assertions produced using a LLM. Notice how, in addition of checking that the deserialized object has an `Integer` value, the exact value of the `Integer` object is also checked. It is worth mentioning that besides this test, FIXCHECK generated another similar test that instantiates the `ExternalTypeWithNonPOJO` object with a `Boolean` object, which also reproduces the same bug in the patch.

III. APPROACH

Figure 2 shows an overview of FIXCHECK our technique for improving patch correctness analysis. FIXCHECK takes as input a patch p, a fault-revealing test t of the bug being addressed by p (i.e., a test that fails on the unpatched version of the program and passes on the patched version), and the failure trace f_t of t. FIXCHECK then attempts to produce a fault revealing test t' for the patch p. FIXCHECK works in three main steps: (1) a *Test Prefixes Generation* step that generates a set of tests that are similar to t, (2) an *Assertion Generation* step that derives meaningful assertions for each test, and (3) a *Failing Tests Selection and Prioritization* step that selects and prioritizes the failing tests based on their likelihood of actually revealing a defect in the patch p.



Test Prefixes Generation (1)
Assertion Generation (2)
public void m(…) {
…
}
patch
@Test
public void inputTest() {
…
assert(…);
}
initial bug revealing test
java.lang.Exception:
at …
at …
at mysuite.inputTest(…)
initial failure trace
Input Transformer
int provider
String provider
…
@Test public void similarTest1() { … }
@Test public void similarTest2() { … }
@Test public void similarTestN() { … }
similar test prefixes without assertions
execution
pass
fail
Assertion Generator
LLM
Failing Tests Selection and Prioritization (3)
Tests Prioritizer
@Test public void similarTestN() { … }
@Test public void similarTest5() { … }
ranked failing tests
java.lang.Exception: at … at … at similarTest1(…)
java.lang.Exception: at … at … at similarTest5(…)
failing tests traces
@Test public void similarTest1() { … assert(…); }
@Test public void similarTest3() { … assert(…); }
similar test prefixes with assertions

Fig. 2: An overview of our approach.

A. *Test Prefixes Generation*

Our approach begins by generating a set of test prefixes (excluding assertions) *similar* to the provided fault-revealing test t. As illustrated in Figure 2, this initial step takes test t and patch p as input, producing a set $\mathcal{T}$ of test prefixes. The parameter n determines the number of tests to be generated. These tests closely resemble test t but differ slightly in input values. To achieve this, n copies of test t are created, with each copy having a concrete input value replaced by a new random value. The decision of which value to change is made randomly. The replacement value is selected from a set of *input providers*—components generating random inputs of specific types. Initially, each provider is instantiated with concrete values obtained from other tests in the same test suite as test t and from the program being patched, obtained through static analysis. To provide a new input value for a given type, the provider randomly selects one of its values or uses a new random value of the same type. Our current implementation includes providers for the primitive types `boolean`, `int`, `long`, `float`, `double`, `char`, and non-primitive types `String` and `Object`. The boolean provider simply returns `true` or `false`, numeric providers offer random values from collected ones or within a configured range [min, max], the char provider alternates between a randomly chosen value or any random character, and the string provider alternates between a randomly chosen string or generates a new string by concatenating a previously collected string with a random character. The object provider uses any value collected for other providers. User-defined type providers can be easily implemented and added.

After generating the set $\mathcal{T}$ of tests, each test is compiled and executed. A failing test during execution is considered a *potential bug* witness in the patch, advancing it to the selection and prioritization step (Section III-C). If the test passes, it moves to the assertion generation step (Section III-B) to generate meaningful assertions. Thus, two sets are obtained at the end of this step: a set $\mathcal{P}$ of passing tests and a set $\mathcal{F}$ of failing tests.

B. *Assertion Generation*

Assertion generation is a pivotal step in our approach, focusing on producing meaningful assertions for the set $\mathcal{P} \subseteq \mathcal{T}$ of passing tests generated in the previous step. To achieve this, we utilize a code language model, `replit-code`[2], specifically designed for code completion. This model, developed and trained by Replit, Inc., is a causal language model focused on code completion and has been trained on a subset of the Stack Dedup v1.2 dataset [23], a 3.1 TB dataset containing permissively licensed source code in 30 programming languages, with Java being the second most represented language.

Despite the availability of other code language models [6], [46], we opted for `replit-code` for two main reasons. Firstly, it is an open-source model, ensuring the replicability of our approach. Secondly, being a model trained for code completion aligns well with the goal of generating assertions.

In the integration of `replit-code` within FIXCHECK, for each passing test $t_p \in \mathcal{P}$ generated in the previous step, we create a prompt to serve as input for the model. Given that replit-code is trained for code completion, the ideal prompt is an incomplete piece of code. To guide the model towards generating assertions, the prompt is crafted by combining the initial fault-revealing test t with the test t_p, lacking assertions. This way, the prompt comprises a sample test with assertions (t) and the test t_p to be completed with assertions. Figure 3 shows the prompt that FIXCHECK would pass to `replit-code` for the test `test0` from our motivating example. The model also requires the provision of a set of parameters, including the maximum number of tokens to be generated during the completion and the *temperature*, which controls the randomness of the generated tokens. We set the maximum number of tokens to 48 since it typically results in the generation of 3 to 4 assertions. The temperature is set to 0.2, the default value. After `replit-code` completes the assertions, we extend the test t_p with the generated assertions, disregarding any additional code that may have been produced.

[2]https://github.com/replit/ReplitLM/tree/main/replit-code-v1-3b

```
// [PROMPT]
public void testWithScalar118() {
  ObjectMapper mapper = new ObjectMapper();
  ExternalTypeWithNonPOJO input = new
     ExternalTypeWithNonPOJO(new Date(123L));
  String json = mapper.writeValueAsString(input);
  assertNotNull(json);
  // and back just to be sure:
  ExternalTypeWithNonPOJO result = mapper.readValue(json,
     ExternalTypeWithNonPOJO.class);
  assertNotNull(result.value);
  assertTrue(result.value instanceof java.util.Date);
}

public void test0() {
  ObjectMapper mapper = new ObjectMapper();
  ExternalTypeWithNonPOJO input = new
     ExternalTypeWithNonPOJO(new Integer(54));
  String json = MAPPER.writeValueAsString(input);
  // and back just to be sure:
  ExternalTypeWithNonPOJO result = MAPPER.readValue(json,
     ExternalTypeWithNonPOJO.class);
--------------------------------------------------------------
// [MODEL RESPONSE]
  assertNotNull(result.value);
  assert(result.value instance of Integer);
  assertEquals(Integer.valueOf(54), result.value);
}
```

Fig. 3: Sample of a prompt used to call the `replit-code` model and the assertions generated by the model.

This process is repeated for every test in $\mathcal{P}$. Each test is compiled and executed as new assertion statement is added. If the execution fails, the test is moved from $\mathcal{P}$ to the set of failing tests $\mathcal{F}$. Otherwise, the test remains in $\mathcal{P}$.

However, it is important to note that assertions generated by `replit-code` model, or any other language model we may use, are not necessarily correct. This may cause FIXCHECK to generate tests that fail due to the presence of incorrect assertions. Nevertheless, since these assertions result in failures unrelated to the initial fault-revealing test, they are considered irrelevant, and FIXCHECK is likely to discard them in the next step.

C. *Failing Tests Selection and Prioritization*

The final phase of our process involves selecting and prioritizing the failing tests within $\mathcal{F}$. Our primary objective is twofold: firstly, to eliminate irrelevant failing tests—those failing due to reasons unlikely to be connected to a defect in the analyzed patch (e.g., failures arising from unexpected inputs); and secondly, to prioritize the remaining failing tests. Prioritization aims to identify and report the tests that are more likely to unveil a bug in the patch.

Given the failure trace f_t (as a sequence of method invocations) of the initial fault revealing test t and the set $\mathcal{F}$ of generated failing tests with their respective failure traces, we first compute a score for each test $t_i \in \mathcal{F}$, based on the similarity between its failure trace f_i and the failure trace f_t. Let f_1 be the failure trace of a failing test $t_1 \in \mathcal{F}$, the similarity score with respect to f_t is based on the Levenshtein distance between the strings corresponding to f_1 and f_t. Essentially, it represents the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into the other. Let s_1 be the string corresponding to f_1 and s_t the string corresponding to f_t, the final score for the generated failing test t_1 is computed as follows:

$$\text{score}(t_1) = \frac{m - \text{Levenshtein}(s_1, s_t)}{m}$$

where m is the maximum length between s_1 and s_t. The more similar the strings s_1 and s_t are, the closer to 1 the score is. Once the scores for each test in $\mathcal{F}$ have been computed, we proceed to discard and prioritize the tests.

Discarding Failing Tests: we discard all tests with a score below a predefined threshold, denoted as K, which is a parameter in our approach. Additionally, if the failure trace of a failing test deviates significantly from the failure trace of the initial fault-revealing test, it is also discarded.

Ranking Failing Tests: the ranking simply sorts the tests in descending order according to their score. If there is more than one failing test with a score $>= K$, the test with the highest score is reported as evidence of the incorrectness of the patch.

Note that in our current prototype the passing tests in $\mathcal{P}$ are not used for further analysis. However, they could be included in the code base to serve as regression tests. Moreover, by incorporating more sophisticated mechanisms during the test input generation, such as fuzzing, one could also focus on increasing the coverage of the patch under analysis, similarly to techniques focused on testing the patch [32].

D. *Implementation*

The current implementation of FIXCHECK is in Java, and supports the analysis of Java programs. Supporting other languages is feasible, although it will demand a considerable effort to implement the test generation step, as the current static analysis, test compilation and execution are tied to Java.

The test generation step uses the JavaParser library [1] to perform the static analysis of the initial fault-revealing test and the target patch, and also to generate the new tests prefixes.

The assertion generation step uses the `replit-code` model via a very simple REST API, implemented in Python with the Flask library [2]. The main reason for this choice is that the model is accessible through HuggingFace[3], which provides a simple interface to access, load and use the model.

It is worth to note that the current implementation of FIXCHECK can be easily extended with other test generation processes and with other assertion generation mechanisms. The tool, as well as all the data and scripts used in our evaluation, are publicly available in our replication package [3].

IV. EVALUATION

To evaluate FIXCHECK, we performed a series of experiments focused on the following research questions:

RQ1 Is FIXCHECK effective in generating fault-revealing tests for incorrect patches?

RQ2 How does FIXCHECK complement with patch correctness assessment techniques?

[3]https://huggingface.co/replit/replit-code-v1-3b

TABLE I: Incorrect patches from BF4J, our dataset comprising developer-written patches in open source projects.

Project	GitHub Repository	Patches
avro	apache/avro	1
choco	chocoteam/choco-solver	1
cli	apache/commons-cli	3
csv	apache/commons-csv	1
graphhopper	graphhopper/graphhopper	3
io	apache/commons-io	1
jackson-databind	FasterXML/jackson-databind	3
jxpath	apache/commons-jxpath	2
lang	apache/commons-lang	1
math	apache/commons-math	2
time4j	MenoData/Time4J	3
Total		21

RQ3 What is the impact of assertions generated by LLMs in the performance of FIXCHECK?

RQ4 How efficient is FIXCHECK?

RQ1 analyzes the effectiveness of FIXCHECK in generating fault-revealing tests for incorrect patches, with respect to a ground truth. RQ2 focuses on evaluating how FIXCHECK can complement the analysis performed by the state-of-the-art patch correctness assessment tools by providing additional evidence of the incorrectness of a patch.

We chose the dynamic technique PATCH-SIM [49] and the hybrid approach Shibboleth [11] because of their superior performance compared to other techniques. We do not consider Opad [52] since it targets C programs, and it is thus out of the scope of our evaluation. Finally, RQ3 evaluates the impact of LLM-generated assertions in the performance of FIXCHECK, while RQ4 is focused on the efficiency of the approach.

A. Dataset

Our evaluation encompasses a total of 160 patches sourced from two distinct origins: a dataset of patches manually crafted by developers in open-source projects and a dataset of patches automatically generated by APR tools. To assess the efficacy of FIXCHECK in generating fault-revealing tests for incorrect patches, a dataset of incorrect patches with known fault-revealing tests is crucial. We specifically refer to the test that unveils the defect in an incorrect patch, not the conventional fault-revealing test for bugs (as in Defects4J [20]). Since no such dataset is readily available, we created our own dataset, named BF4J. This dataset comprises 40 incorrect patches manually produced by developers during the maintenance of various open-source projects. We selected 11 popular projects from GitHub, including 6 from Defects4J, and collected the patches by automatically mining issues using the GitHub API. Filtering issues exhibiting the *commit*, *close*, *reopen* pattern, we obtained 26,937 issues, of which 249 showed the pattern of interest. Through manual analysis, we confirmed that 209 issues were unrelated to bug fixes or were closed after discussion, and thus, they were discarded. The remaining 40 issues underwent careful analysis to identify the incorrect patches, resulting in the 40 patches in our dataset. For our experiments, we focused on analyzing 21 patches, as these included subsequent commits containing fault-revealing tests needed as input by FIXCHECK. Table I displays the distribution of the incorrect patches from BF4J used in our experiments, while the complete dataset is available in our replication package [3].

TABLE II: Dataset of patches generated by APR tools.

Project	Patches	Correct	Incorrect
chart	26	3	23
lang	15	5	10
math	83	20	63
time	15	2	13
Total	139	30	109

To evaluate how FIXCHECK can complement the analysis performed by patch correctness assessment tools, we consider a dataset of patches for which the correctness has been previously assessed. This dataset was collected and used in the evaluation of PATCH-SIM [49], and later used in the evaluation of Shibboleth [11]. It contains 139 patches, with 30 being correct and 109 being incorrect. The patches were generated by the APR tools jGenProg [34], Nopol [51] (versions 2015 and 2017), jKali [34], ACS [50] and HDRepair [24]. Within these tools, jGenProg and jKali are Java implementations of the search-based tools GenProg [13] and Kali [40], respectively; Nopol is a tool relying on constraint solving; and ACS and HDRepair are based on statistical models. All the patches were generated while addressing bugs in the Defects4J. Precisely, a total of 74 bugs were analyzed, including bugs from the following projects: chart, lang, math and time. Table II shows the distribution of patches per project.

B. Experimental Setup

FIXCHECK requires as input a potentially incorrect patch and the initial fault-revealing test (of the bug being addressed) with its corresponding failure trace. Thus, before analyzing each patch, we collect the initial fault-revealing tests with their traces. To do so, we build each project on the buggy version, and then run the fault-revealing test collecting the trace. In the case of the 139 patches generated by the APR tools, Defects4J already includes the initial fault-revealing test for each bug, and also provides the infrastructure to build the projects and run the tests. For the 21 patches from our dataset, similarly to Defects4J, we implemented a framework to automate the process of building the versions related to each patch. Thus, for these patches we use our framework to build the buggy version and then run the initial fault-revealing test.

Once we have the initial fault-revealing test and its trace for each patch, we apply the patch to the buggy version, and build the project. From this point, we execute FIXCHECK on each patch. In all our experiments we instruct FIXCHECK to generate 100 new tests during the test generation step. During the assertion generation step, `replit-code` is configured to generate 48 tokens, which results on an average of 3 to 4 assertions per test. In the final selection and prioritization step, we configure the parameter $K = 0.4$. To account for the

TABLE III: Effectiveness of FixCheck generating fault-revealing tests for incorrect patches.

Project	#P	Failing Tests			Fault-revealing Tests			
		Total	Discarded	Ranked	#Reported	Correct	Precision	Recall
avro	1	49	49	0	0	0	1.0	0.0
choco	1	55	55	0	0	0	1.0	0.0
cli	3	133	14	119	2	2	1.0	0.67
csv	1	5	0	5	1	1	1.0	1.0
graphhopper	3	176	114	62	2	2	1.0	0.67
io	1	0	0	0	0	0	1.0	0.0
jackson-databind	3	288	213	75	1	1	1.0	0.33
jxpath	2	151	42	109	2	2	1.0	1.0
lang	1	98	98	0	0	0	1.0	0.0
math	2	115	19	96	2	2	1.0	1.0
time4j	3	268	115	153	3	3	1.0	1.0
Total	21	1,338	719	619	13	13	1.0	0.62

randomness of our approach, we run FixCheck a total of 10 times on each patch, and then report the average results.

All the experiments were run on a workstation with a Xeon Gold 6154 CPU (3GHz), with 128 GB of RAM, running Debian/GNU Linux 11. Detailed instructions on how to replicate the experiments are available in the replication package [3].

C. *Effectiveness of* FixCheck *(RQ1)*

The effectiveness of FixCheck to generate fault-revealing tests for incorrect patches is measured against the ground truth. The experiment consisted in running FixCheck on each incorrect patch in our dataset BF4J, and then measuring the number of patches for which FixCheck generated a fault-revealing test reproducing the same problem as the fault-revealing test in the ground truth. Concretely, we measure the precision and recall as follows. Let TP (true positives) be the number of reported fault-revealing tests that are correct, FP (false positives) be the number of reported fault-revealing tests that are incorrect, and FN (false negatives) be the number of incorrect patches for which no fault-revealing test was generated. Then, precision and recall are defined as follows:

$$Precision = TP/(TP + FP) \quad Recall = TP/(TP + FN)$$

Intuitively, precision measures the proportion of reported fault-revealing tests that are actually correct, while recall measures the proportion of incorrect patches for which FixCheck is able to generate a fault-revealing test. To confirm the correctness of a generated fault-revealing test, we conduct a careful manual inspection to determine whether it failed for the same reason as the test in the ground truth.

FixCheck was executed as described earlier in this section. Table III summarizes the results of this experiment, grouped by project. The column #P shows the number of incorrect patches on each project. The column Failing Tests groups the number of failing tests generated, including the number of discarded and ranked tests. Finally, the column Fault-revealing tests shows the number of tests reported as being fault-revealing, the number of correct fault-revealing tests, and the corresponding precision and recall values.

Precision: FixCheck achieves perfect precision with $K = 0.4$, i.e., the 13 reported fault-revealing tests are correct. Of course, precision is affected by the value of our parameter K used to decide weather a fault-revealing is ranked or not. However, still for small values of K, the precision is still high (~70%), as shown in Table IV, which shows how the precision and recall of FixCheck vary as the value of K is increased.

TABLE IV: Effect of the parameter K on the effectiveness of FixCheck.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
prec.	0.68	0.72	0.87	1.0	1.0	1.0	1.0	1.0	1.0	1.0
rec.	0.62	0.62	0.62	0.62	0.57	0.48	0.33	0.1	0.0	0.0

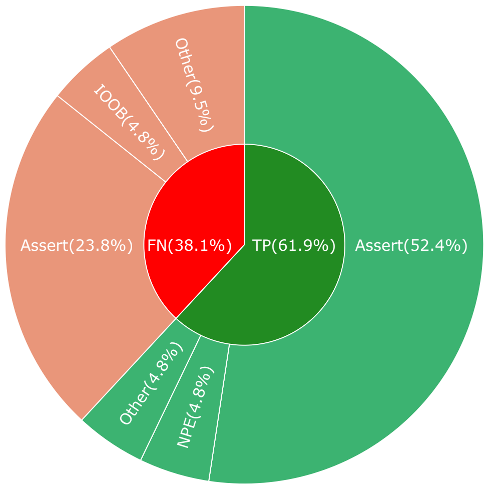


Fig. 4: Breakdown of failure reasons of the fault-revealing tests in the ground truth and how FixCheck performed on them.

Recall: FixCheck achieves a recall of ~62%, generating a correct fault-revealing test for 13 out of the 21 incorrect patches. To better understand the cases for which FixCheck can and cannot generate a fault-revealing test, in Figure 4 we show a breakdown of the failure reasons of the initial fault-revealing tests of each patch in the ground truth, and how FixCheck performed on them. Regarding the 13 patches for which FixCheck generated a correct fault-revealing test, 11 of them (~52%) required an assertion, while the other 2 patches required just to include a specific input value in the test. This gives relevance to the importance of having an effective assertion generation technique. Regarding the

TABLE V: FIXCHECK complementing the patch correctness analysis techniques PATCH-SIM and Shibboleth.

Project	**#IP**	**PATCH-SIM**		**Shibboleth**	
		#Detected	**#FR-Tests(%)**	**#Detected**	**#FR-Tests(%)**
chart	23	13	7 (53.8%)	21	10 (47.6%)
lang	10	5	2 (40.0%)	10	6 (60.0%)
math	63	35	17 (48.6%)	58	28 (48.2%)
time	13	9	5 (55.6%)	12	5 (41.6%)
Total	109	62	31 (50.0%)	101	49 (48.5%)

8 patches for which FIXCHECK was not able to generate a fault-revealing test, in 5 of them (∼24%) it failed to produce the right assertion, while in the remaining 3 cases it failed to generate a test using a specific input value to trigger exceptions of type `InvalidAvroMagicException`, `StringIndexOutOfBoundsException` and `RuntimeException`.

D. Complementing Patch Correctness Assessment (RQ2)

One of the goals of FIXCHECK is to complement existing patch correctness assessment techniques, by providing further evidence of the incorrectness of a patch. In this experiment we evaluate how FIXCHECK complements two state-of-the-art techniques: PATCH-SIM [49] and Shibboleth [11]. The performance of these techniques was previously evaluated in patches generated by APR tools while addressing bugs in Defects4J [20]. Although both techniques, notably Shibboleth, were able to detect a significant number of incorrect patches, with a high precision, they only report the verdict of the correctness, without providing any additional information.

To evaluate how our approach can complement these techniques, and to what extent it can improve the output of the analysis, we execute FIXCHECK on all the patches that were classified as incorrect by PATCH-SIM and Shibboleth, and measured for how many of them FIXCHECK is able to generate a fault-revealing test. The results of this experiment are shown in Table V, separated by project. The column #IP shows the number of incorrect patches, the column #Detected shows the number of patches detected by the corresponding technique, and the column FR-Tests shows the number of patches for which FIXCHECK was able to generate a fault-revealing test.

PATCH-SIM: PATCH-SIM, the fully dynamic technique, detects 62 out of the 109 incorrect patches (a 56.8% of the total). FIXCHECK is able to generate at least one fault-revealing test for 50% of these patches. Moreover, we executed FIXCHECK in the incorrect patches missed by PATCH-SIM, and FIXCHECK generated fault-revealing tests for 18 of them.

Shibboleth: Shibboleth, a more recent hybrid tool, is able to detect 101 of them (a 92.6% of the total). For these, FIXCHECK is able to generate at least one fault-revealing test for 48.5% of them.

Overall, considering both techniques, FIXCHECK can generate a fault-revealing test for around 50% of the detected incorrect patches. This indicates that FIXCHECK can effectively complement the analyses of these techniques by providing useful information regarding the incorrectness of a patch in the form of a new fault-revealing test, that could be used by the developers to debug and improve their patches.

TABLE VI: FIXCHECK's performance under different assertion generation mechanisms: w.r.t the ground truth (as in RQ1) and complementing patch correctness assessment (as in RQ2).

Mechanism	**Effectiveness**		**Patches with FR-Tests (%)**	
	Precision	**Recall**	**PATCH-SIM**	**Shibboleth**
no-assertion	0.86	0.29	19.4	22.0
previous	0.53	0.42	62.9	59.6
replit-code	1.0	0.62	50.0	48.5

The dataset of this experiment also contains 30 correct patches. For these, FIXCHECK wrongly generated a "fault revealing test" for 10 of them. However, as FIXCHECK is intended to be used in conjunction with a patch correctness assessment technique that can provide certain confidence on the incorrectness of a patch, this is not a major concern.

E. Impact of LLM-generated assertions (RQ3)

To analyze the impact of the LLM-generated assertions we compare three different assertion generation mechanisms: (1) no assertions, (2) reusing the assertions present in the initial fault-revealing test, and (3) using `replit-code` as assertion generator. For each mechanism, we measure the effectiveness of FIXCHECK in generating fault-revealing tests w.r.t to the ground truth (as in RQ1), and the percentage of incorrect patches identified by patch correctness assessment tools for which FIXCHECK can generate a fault-revealing test (as in RQ2). The results of this analysis are shown in Table VI.

No assertions: without assertions FIXCHECK generates fault-revealing tests for a lower number of incorrect patches, resulting in a low recall when analyzing effectiveness (29%) and a lower performance when complementing patch correctness assessment techniques, generating fault-revealing tests for no more than 22% of the patches. Nevertheless, FIXCHECK's tests (without assertions) can still be useful.

Previous assertions: although using assertions present in the initial fault-revealing test can improve in terms of recall (42%), precision is significantly reduced (53%). The problem is that reusing assertions is not always adequate, as it can result in failing tests with assertions unrelated to the actual fault. Our motivating example (Section II) reflects this. Although the percentage of incorrect patches for which a fault-revealing test is generated when reusing assertions, the percentage is the highest among the three mechanisms, it is very likely that such tests contains failing assertions unrelated to the actual fault, as the precision of FIXCHECK is significantly reduced.

LLM-generated assertions: using the assertions generated by `replit-code` is the best option, as they allow to effectively improve the output of patch correctness assessment techniques (generating fault-revealing tests for up to 50% of the detected incorrect patches), while performing better than the other mechanisms in terms of precision and recall. This indicates that the assertions generated by `replit-code` are

TABLE VII: Analysis time of FIXCHECK.

Project	#P	Time (sec.)			
		Test Gen	Assert. Gen	Test Exec.	All
chart	26	0	148,242	183	148,425
lang	15	3	79,437	1,025	80,492
math	83	1	469,677	2,831	472,509
time	15	1	52,251	125	52,377
BF4J	21	0	40,312	68	40,380
Total	160	5	789,919	4,259	794,183

more likely to be consistent with the test in which they are placed. In fact, we observed that the model generates assertions that are used in the initial fault-revealing test when appropriate, thus "subsuming" the previous mechanism.

Note that FIXCHECK is independent of the used LLM. Indeed, we experimented with `Llama2` [44], recently open sourced by Meta. When using the 13B parameter model, FIXCHECK achieves the same effectiveness as with `replit-code`, and better performance complementing patch assessment techniques (generating fault-revealing tests for up to 53.2% of the incorrect patches). Given the increasing availability of LLMs, this is a very positive aspect of FIXCHECK.

F. *Efficiency of* FIXCHECK *(RQ4)*

For each patch we measure the time spent during the most relevant steps of FIXCHECK's analysis process: test generation, assertion generation, and test execution. The results of this analysis are shown in Table VII, where time is measured in seconds.

Notably, the results show that FIXCHECK spent only 5 seconds to generate the 16,000 tests (100 per analyzed patch), which evidences the efficiency of our static analysis-based test generation. This indicates that we could instruct FIXCHECK to generate many more tests without a significant overhead.

In our experiments the most time-consuming step is the assertion generation step. It took a total of 789,919 seconds, with an average of 4,906 seconds ($\sim$81 minutes) per patch, and around 1 minute per single test. Note that this does not have to do with FIXCHECK's approach, but with the fact that we use a single CPU workstation to perform our experiments, which negatively affects the performance of `replit-code`. It is well-known that for LLMs to be efficient, better computational resources are required, including GPUs. In fact, we have observed that the we can generate assertions in a few seconds per test when using other available models such as Codex [6], available through the OpenAI API. As mentioned previously, we favor the use of `replit-code` since it is a lightweight open source model that can be easily used.

Finally, FIXCHECK spent a total of 4,259 seconds ($\sim$71 minutes) executing the generated tests, which is a reasonable time considering that for each single patch the average time spent executing the 100 generated tests is around 26 seconds.

V. LIMITATIONS

The limitations of FIXCHECK's approach are mainly in its test generation capabilities. Currently, the only transformations supported are input transformations, which means that FIXCHECK can generate new tests just by replacing the input values of the initial fault-revealing test. This may limit the ability of FIXCHECK to generate fault-revealing tests that would require generating a test that differs from the initial fault-revealing test in more than just the input values. In fact, for 16 out of the total 40 incorrect patches present in BF4J, we identified that the test that developers provided to reproduce a defect in the patch, also required to remove/add a method call with respect to the initial fault-revealing test.

FIXCHECK is also limited in the assertion generation process, as it only adds assertions at the end of the test, ignoring those that may be generated for intermediate statements in the test sequence. This limitation could be addressed by using a more sophisticated LLM model that can be instructed to re-generate the entire test including assertions, instead of just letting the model complete the test sequence.

VI. THREATS TO VALIDITY

Our evaluation uses 21 patches from BF4J, a meticulously studied dataset of incorrect patches collected from open-source projects. To minimize errors, we executed the tests provided in posterior commits that revealed the defect in the patch, to ensure they failed. We publicly release our collected dataset for review by the community.

Additionally, our evaluation of FIXCHECK's effectiveness to generate fault-revealing tests for incorrect patches involved a manual analysis of the generated tests. In this case, we compared each reported failing test with the fault-revealing test from the ground truth, to ensure that the generated test was indeed able to reproduce the defect in the patch. All generated tests are also publicly available.

Threats to internal validity may arise from the randomness involved in the FIXCHECK's test generation and assertion generation processes. To account for this threat, we analyzed FIXCHECK's executions over multiple runs, and reported the average results. Our current evaluation is limited to 160 patches, which is a relatively small number compared to other recent studies on patch correctness [11], [43]. In the future, we plan to extend the experimental evaluation to bigger datasets.

VII. RELATED WORK

Automated Program Repair. APR techniques aim to automate the generation of patches for buggy programs, relieving developers from the manual fixing process. Recent approaches have shown promising results in bug fixing [14], [28]. The majority of APR techniques adopt the generate-and-validate approach, wherein a set of candidate patches is generated and subsequently validated for adequacy through static analysis or test suites [40]. Despite their efficacy, many APR techniques may still produce incorrect patches that overly tailor to test suites without effectively addressing the bugs [40], [41]. This work has the potential to complement existing efforts by automatically generating test cases that reveal the incorrectness of a patch, offering a means to identify potentially overfit patches generated by APR tools.

Patch Correctness Assessment. To address the overfitting challenges of APR techniques, various effective approaches for assessing the correctness of generated patches have been proposed. Many of these approaches adopt dynamic methodologies, requiring the execution of the patched program to analyze its behavior. PATCH-SIM [49], a state-of-the-art dynamic approach, aims to identify overfitting patches by quantifying behavioral changes between the buggy program and the patch. It utilizes a test generation tool (Randoop [39]) to automatically generate tests that measure these behavioral differences, subsequently heuristically discarding patches with significantly different behavior from the original program. DiffTGen [48] focuses on patches generated by APR tools, aiding researchers in classifying patches more confidently. It uses test generation (EvoSuite [9]) to generate new test inputs that reveal semantic differences between the original faulty program and the patched program. However, it requires the correct patch as an oracle for classification, making it less practical for developers. Similarly, Invalidator [25], based on semantic and syntactic analysis, also relies on the correct patch for its semantic reasoning, limiting its practicality. Opad [52] uses fuzz testing to provide new test inputs for the buggy program, employing two predetermined oracles to detect patches introducing crash or memory-safety problems. xTestCluster [33] is a test-based patch clustering approach that minimizes the number of plausible patches for developers to review. It analyzes patches generated by repair tools, leveraging information from the execution of newly generated test cases to cluster patches. All these dynamic techniques utilizing test generation rely on specific test generation tools as components in their analysis processes. In contrast, FIXCHECK implements its test generation process based on static analysis, offering a more lightweight approach focused on generating new test cases that differ slightly from the initial fault-revealing test.

Static and hybrid approaches have also been proposed. Tan et al. [42] introduced a static technique using anti-patterns to avoid incorrect patches, but its accuracy has been questioned [45]. BATS [43], a recent static approach similar to FIXCHECK, leverages the initial fault-revealing test. For a given failing test case, BATS computes similarity metrics to identify historical similar test cases, aiming to identify associated applied patches. Shibboleth [11], the most recent hybrid approach, outperforms static and dynamic approaches by measuring syntactic and semantic changes, as well as code coverage changes between the buggy program and plausible patches. Despite existing techniques for patch assessment, such as Evocatio [19], Opad [52], and Katch [32], which primarily focus on predicting the correctness of patches generated by APR tools, FIXCHECK stands out by offering a unique capability: the automatic generation of test cases that serve as additional evidence of the incorrectness of a patch, complementing these existing approaches.

Large Language Models. LLMs have shown a promising performance in tasks related to code generation. Various works have studied the use of code language models (specifically trained on code and on code-related tasks) to improve APR [8], [18], [47]. CodaMosa [26] employs LLMs to improve code coverage in search-based software testing techniques. Kang et al. [21] use LLMs to, given a faulty program and a bug report, produce the failing test cases that reproduces the bug. Non-LLM based approaches have also been proposed for test completion [38], based on deep learning models using code semantics. We leverage the power of LLMs to generate meaningful assertions for test cases, which, to the best of our knowledge, is their first use for this purpose.

VIII. Conclusion and Future Work

The assessment of patch correctness is a challenging task with significant implications for the effectiveness of APR tools and the potential to aid developers in the patch development process. Although effective techniques for patch correctness assessment exist, their output often provides a binary classification of patches as either correct or incorrect. To overcome this limitation and offer insights into the nature of incorrect patches, we introduced FIXCHECK—a technique that combines static analysis, random testing, and large language models to generate fault-revealing tests for incorrect patches. Our evaluation demonstrates that FIXCHECK is capable of generating fault-revealing tests, showcasing the incorrectness of 62% of patches authored by developers. Moreover, we illustrate that FIXCHECK enhances the output of patch correctness assessment techniques by generating fault-revealing tests for up to 50% of detected incorrect patches. While our work exhibits promising results, we recognize several limitations in our approach that present avenues for improvement. The test generation process could benefit from additional transformations, such as adding or removing statements, to generate more diverse tests. The random testing-based input selection may be enhanced by incorporating alternative techniques like fuzzing. Our assertion generation process, currently adding assertions only at the end of tests, might be further refined to consider assertions for intermediate statements in the test sequence. The modular structure of our technique allows for targeted improvements in specific components, such as enhancing test generation diversity or incorporating more precise assertions in the assertion generation process. Future extensions of our tool will address these limitations and expand our evaluation to include a broader range of patches. Additionally, we plan to investigate how fault-revealing tests generated by FIXCHECK can be effectively utilized by developers or APR tools to enhance the patch generation process.

Acknowledgments

This work was funded by the Madrid regional government program S2018/TCS-4339 (BLOQUES-CM) and by the Spanish Governement MCIN/AEI/10.13039/501100011033/ERDF through grants TED2021-132464B-I00 (PRODIGY) and PID2022-142290OB-I00 (ESPADA). Those projects are cofunded by European Union ESF, EIE, and NextGeneration funds.

References

[1] Javaparser. https://javaparser.org/, 2019.

[2] Flask. https://flask.palletsprojects.com/, 2022.

[3] FIXCHECK implementation and replication package. https://zenodo.org/records/10498174, 2024.

[4] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA):159:1–159:27, 2019.

[5] Liushan Chen, Yu Pei, and Carlo A. Furia. Contract-based program repair without the contracts. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 637–647. IEEE Computer Society, 2017.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[7] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. Software Eng.*, 47(9):1943–1959, 2021.

[8] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *Proceedings of the 45th International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1469–1481. ACM, 2023.

[9] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE*, pages 416–419. ACM, 2011.

[10] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 19–30. ACM, 2019.

[11] Ali Ghanbari and Andrian Marcus. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 654–665. ACM, 2022.

[12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 3–13. IEEE Computer Society, 2012.

[13] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[14] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, 2019.

[15] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. Automatic program repair. *IEEE Softw.*, 38(4):22–27, 2021.

[16] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. Has the bug really been fixed? In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 55–64. ACM, 2010.

[17] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 12–23. ACM, 2018.

[18] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. In *Proceedings of the 45th International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. ACM, 2023.

[19] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. Evocatio: Conjuring bug capabilities from a single poc. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1599–1613. ACM, 2022.

[20] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM, 2014.

[21] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction, 2023.

[22] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search (T). In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 295–306. IEEE Computer Society, 2015.

[23] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code, 2022.

[24] Xuan-Bach Dinh Le, David Lo, and Claire Le Goues. History driven program repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 213–224. IEEE Computer Society, 2016.

[25] Thanh Le-Cong, Duc-Minh Luong, Xuan-Bach Dinh Le, David Lo, Nhat-Hoa Tran, Bui Quang Huy, and Quyet-Thang Huynh. Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning. *IEEE Trans. Software Eng.*, 49(6):3411–3429, 2023.

[26] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931, 2023.

[27] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: revisiting template-based automated program repair. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 31–42. ACM, 2019.

[28] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. A critical review on the evaluation of automated program repair systems. *J. Syst. Softw.*, 171:110817, 2021.

[29] Fan Long and Martin C. Rinard. Staged program repair with condition synthesis. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178. ACM, 2015.

[30] Fan Long and Martin C. Rinard. Automatic patch generation by learning correct code. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 298–312. ACM, 2016.

[31] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. Sapfix: automated end-to-end repair at scale. In Helen Sharp and Mike Whalen, editors, *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 269–278. IEEE / ACM, 2019.

[32] Paul Dan Marinescu and Cristian Cadar. KATCH: high-coverage testing of software patches. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering*

Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pages 235–245. ACM, 2013.

[33] Matias Martinez, Maria Kechagia, Anjana Perera, Justyna Petke, Federica Sarro, and Aldeida Aleti. Test-based patch clustering for automatically-generated patches assessment, 2022.

[34] Matias Martinez and Martin Monperrus. ASTOR: a program repair library for java (demo). In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 441–444. ACM, 2016.

[35] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 129–139. ACM, 2018.

[36] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 448–458. IEEE Computer Society, 2015.

[37] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 772–781. IEEE Computer Society, 2013.

[38] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Learning deep semantics for test completion. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2111–2123. IEEE, 2023.

[39] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84. IEEE Computer Society, 2007.

[40] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 24–36. ACM, 2015.

[41] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 532–543. ACM, 2015.

[42] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 727–738. ACM, 2016.

[43] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kaboré, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. Predicting patch correctness based on the similarity of failing test cases. *ACM Trans. Softw. Eng. Methodol.*, 31(4):77:1–77:30, 2022.

[44] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[45] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. Automated patch correctness assessment: How far are we? In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 968–980. IEEE, 2020.

[46] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics, 2021.

[47] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. ACM, 2023.

[48] Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 226–236. ACM, 2017.

[49] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 789–799. ACM, 2018.

[50] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 416–426. IEEE / ACM, 2017.

[51] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Trans. Software Eng.*, 43(1):34–55, 2017.

[52] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 831–841. ACM, 2017.