Invalidator: Automated Patch Correctness Assessment via Semantic and Syntactic Reasoning

Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D. Le, David Lo,
Nhat-Hoa Tran, Bui Quang-Huy and Quyet-Thang Huynh

Abstract—Automated program repair (APR) has been gaining ground recently. However, a significant challenge that still remains is patch overfitting, in which APR-generated patches plausibly pass the validation test suite but fail to generalize. A common practice to assess the correctness of APR-generated patches is to judge whether they are equivalent to ground-truth, i.e., developer-written patches, by either generating additional test cases or employing human manual inspections. The former often requires the generation of at least one test witnessing the behavioral differences between the APR-patched and developer-patched programs. Searching for the witnessing test, however, can be difficult as the search space can be enormous. Meanwhile, the latter is prone to human biases and requires repetitive and expensive manual effort.
In this paper, we propose a novel technique, namely INVALIDATOR, to automatically assess the correctness of APR-generated patches via semantic and syntactic reasoning. INVALIDATOR reasons about program semantic via program invariants while it also captures program syntax via language semantic learned from large code corpus using the pre-trained language model. Given a buggy program and the developer-patched program, INVALIDATOR infers likely invariants on both programs. Then, INVALIDATOR determines that a APR-generated patch overfits if: (1) it violates correct specifications or (2) maintains errors behaviors of the original buggy program. In case our approach fails to determine an overfitting patch based on invariants, INVALIDATOR utilizes a trained model from labeled patches to assess patch correctness based on program syntax. The benefit of INVALIDATOR is three-fold. First, INVALIDATOR is able to leverage both semantic and syntactic reasoning to enhance its discriminant capability. Second, INVALIDATOR does not require new test cases to be generated but instead only relies on the current test suite and uses invariant inference to generalize the behaviors of a program. Third, INVALIDATOR is fully automated. We have conducted our experiments on a dataset of 885 patches generated on real-world programs in Defects4J. Experiment results show that INVALIDATOR correctly classified 79% overfitting patches, accounting for 23% more overfitting patches being detected by the best baseline. INVALIDATOR also substantially outperforms the best baselines by 14% and 19% in terms of Accuracy and F-Measure, respectively.

Index Terms—Automated Patch Correctness Assessment, Overfitting problem, Automated Program Repair, Program Invariants, Code Representations

1 INTRODUCTION

Automated program repair (APR) is a promising approach to alleviate the onerous burden on developers to manually fix bugs. A substantial number of APR techniques have been proposed over the years [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], with several breakthroughs that inspired potential practical adoption of APR. Notably, Facebook has recently deployed SapFix [11], the first-ever industrial-scale automatic bug fixing system, for suggesting fixes to developers in real-world products. Despite the recent success, APR still suffers from a major challenge, namely *patch overfitting* [12], [13], in which a generated patch may pass all test cases but still fails to generalize to the intended behaviors of the program. Qi et al. [2] reported that 98% of the plausible patches generated by GenProg [14] are overfitting.

Detecting overfitting patches is one key challenge that needs to be addressed to not only allow for fair comparisons between APR techniques but also enable a practical adoption of APR by developers. So often, one APR technique claims to be better than others only in terms of the number of bugs that it can generate "correct" patches for. Furthermore, recent work suggested that low-quality patches may adversely affect developers' performance [15]. A fundamental question is then,

"How do we determine whether a patch is correct?"

Unfortunately, even with the presence of the ground truth (developer-patched) program, it is difficult to tell if a APR-patched program is correct. That is unless the APR-patched program and the ground truth program are exactly syntactically the same, determining whether two programs are semantically equivalent is indeed an undecidable problem [16].

Recent approaches in APR explored different ways to

- *Thanh Le-Cong, Xuan-Bach D. Le are with School of Computing and Information Systems, The University of Melbourne*
  E-mail: congthanh.le@student.unimelb.edu.au, bach.le@unimelb.edu.au,
- *Duc-Minh Luong, Quang-Huy Bui, Nhat-Hoa Tran and Quyet-Thang Huynh are with School of Information and Communication Technology, Hanoi University of Science and Technology*
  E-mail: {minh.ld176821, huy.bq20202517M} @sis.hust.edu.vn, {hoatnt, thanghq}@soict.hust.edu.vn
- *David Lo is with School of Computing and Information Systems, Singapore Management University*
  E-mail: davidlo@smu.edu.sg
- *Quyet-Thang Huynh is the corresponding author.*

assess the correctness of APR-generated patches concerning available developer-written patches as ground truth. These approaches either use test-suite augmentation or human manual inspections. While being effective [17], the human manual inspections are prone to human biases, are expensive, and require manual, repetitive tasks. The test-suite augmentation approaches, e.g., [18], [7] are fully automated but require generating at least one test case to witness behavioral differences between the APR-patched and ground truth programs. A recent study [17] showed that the test-suite augmentation approaches are indeed ineffective as the search space for bug-witnessing test cases can be large. Even the state-of-the-art test case generation techniques such as Randoop [19] and DIFFTGEN [18] can only identify 22% of the overfitting patches generated by APR tools [17]. Xin et al. [18] also reported that state-of-the-art test generator EVOSUITE [20], in many cases, failed to generate any test methods that exercise code changes introduced in generated patches and thus failed to identify behavioral differences between the patches and the ground truth.

In this paper, we introduce a novel technique, named INVALIDATOR, that incorporates both semantic and syntactic reasoning to automatically assess the correctness of generated patches from APR techniques. INVALIDATOR reasons about *program semantic* via *program invariants*. Simultaneously, it reasons about *program syntax* via *language semantic* learned from large code corpus using pretrained language models. Similar to existing automated patch correctness assessment techniques, INVALIDATOR relies on behavioral differences between the APR-patched and ground truth programs to judge patch correctness. However, conceptually, Invalidator is different from the strategy employed by existing APAC techniques (e.g. DIFFTGEN [18], PATCHSIM [21], or RANDOOP [19], [17]). These techniques generate new tests to augment the current test suite, in which each test generates one execution. Thus, the chance to hit an execution that reveals a behavioral difference between the APR-patched and ground truth programs is approximately linearly proportional to the number of tests generated. In contrast, Invalidator only uses the current test suite and infers program invariants that naturally generalize beyond the test suite. The generalization of program invariants allows Invalidator to effectively and semantically reason about program correctness. Additionally, Invalidator further augments program semantic reasoning by syntactic reasoning to enhance its effectiveness. We describe the details of semantic and syntactic reasonings in Invalidator below.

Given a APR-generated patch, the original buggy program and its correct (ground-truth) version, INVALIDATOR works in two main phases.

① **Semantic-based Classifier.** The semantic-based classifier is built based on two high-level intuitions. First, program invariants that are maintained in both the buggy and correct (ground truth) versions of a program can serve as *correct* specifications of the program. Second, program invariants that only exist in the buggy program but do not exist in the correct version may represent *error* specifications of the program. INVALIDATOR determines that a machine-generated patch overfits if the machine-patched program: (1) violates correct specifications or (2) maintains error specifications. Particularly, INVALIDATOR first automatically infers likely invariants of each program based on its original test suite by using DAIKON [22], a well-known invariant inference tool. INVALIDATOR then constructs the set of *correct and error specifications*, which serve as approximate specifications for the program under test. Based on the inferred specifications, INVALIDATOR determines that a patch is overfitting if invariants inferred from the machine-patched program either violate the correct specifications or maintain error specifications.

② **Syntactic-based Classifier.** In case the invariant-based specification inference fails to determine an overfitting patch, INVALIDATOR further detects overfitting patches via language semantic differences between the machine-generated patch and its buggy and correct version. Specifically, INVALIDATOR employs a pre-trained language model, i.e. CODEBERT to extract source syntactic features from source code of each program. INVALIDATOR then measures the differences by a set of comparison functions, e.g., subtraction or similarity. Finally, INVALIDATOR uses a trained model from labeled data to estimate the likelihood of the machine-generated patch being overfitting based on the syntactic proximity.

We have conducted our experiments on a dataset of 885 patches which include 508 overfitting patches and 377 correct ones generated for large real-world programs in the Defects4J dataset. To investigate the effectiveness of our approach, we compared INVALIDATOR against the state-of-the-art APAC techniques, consisting of RGT [43], ODS [23], BERT+LR [24], PATCHSIM [21], DIFFTGEN [18], ANTI-PATTERNS [25], DAIKON [26]. Experiment results showed that INVALIDATOR correctly classified 79% overfitting patches, accounting for 23% more overfitting patches being detected as compared to the best baseline. Invalidator also remarkably outperforms the best baselines by 14% (0.81 vs. 0.68) and 19% (0.87 vs. 0.76) in terms of *Accuracy* and *F-Measure*, respectively.

In summary, we made the following contributions:

- We introduced INVALIDATOR, a novel technique that uses both semantic reasoning (via program invariants) and syntactic reasoning (via source code features) to automatically validate APR-generated patches. Our empirical evaluation demonstrated that our approach effectively detects 79% overfitting patches with a precision of 97%.
- We introduced two overfitting rules based on program invariants to validate machine-generated patches. Our empirical evaluation shows that these overfitting rules can detect 51 % overfitting patches with a precision of 97%.
- We proposed to use syntactic reasoning from program source code to augment semantic reasoning from two aforementioned overfitting rules. Our empirical evaluation shows that syntactic reasoning can boost the performance of our approach by 35% and 30% in terms of Accuracy and F-Measure, respectively.
- We conducted experiments on 885 machine-generated patches for the Defects4J benchmark. Experiment results show that the unique combination of syntactic and semantic reasoning empowers

INVALIDATOR to achieve substantial improvements (i.e., 19% and 14% for Accuracy and F-Measure, respectively) over state-of-the-art baselines.

The remainder of this paper is structured as follows. Section 2 introduces background on the overfitting problem, APAC, and likely invariants. Section 3 describes a motivating example for our approach, followed by Section 4 that presents our approach in detail. Section 5 describes our experimental setup and results. Section 6 and Section 7 discuss threats to validity and related work, respectively. Finally, Section 8 concludes and presents future work.

2 BACKGROUND

This section presents an outline of recent automated program repair (APR) techniques, the overfitting problem in APR, and techniques for assessing the correctness of APR-generated patches. We subsequently explain program invariants and dynamic invariant detection.

2.1 Automated Program Repair

Program Repair. Given a buggy program and a set of test cases in which there exist at least one failing test, the overall goal of automated program repair (APR) techniques is to generate a patch that passes all the test cases while not introducing new bugs. Generally, APR techniques can be categorized into two main families, including search-based repair and semantic-based repair. Search-based techniques often use meta-heuristic algorithms, e.g. genetic programming [14], random search [27], or learning algorithms such as data mining and machine learning, e.g., [3], [28], [29], and [7], to apply mutations and evolve the buggy program until they find a patch passing the test suite. Semantics-based repair techniques, e.g. ANGELIX [6], S3 [8], JFIX [30], use semantic analysis, e.g. symbolic execution, and program synthesis to construct patches that satisfy certain semantic constraints. We will elaborate in detail on these techniques in the related work section (Section 7).

Overfitting. One key challenge in APR is that APR can generate tests-adequate patches that may not generalize. This is known as the *patch overfitting* problem, in which the generated patches may pass all tests but still are incorrect [2], [12]. Early APR techniques use an existing test suite as an oracle to validate generated patches [31], [14]. Specifically, a patch is considered as correct if it passes all test cases and incorrect otherwise [31], [14]. Recent studies [2], [12] showed that this assessment method is ineffective due to the incompleteness of the test suite used for assessment. By manual analyses, Qi et. al. [2] have shown that the majority of patches generated by search-based APR techniques such as GENPROG [14], AE [32], and RSREPAIR [27] are overfitting. By automated assessment, in which a new independent test suite is generated to cross-validate APR-generated patches, Le et al. [13] also have reported that semantics-based repair techniques, such as ANGELIX [6] and the likes, are no exception to the overfitting issue.

Automated Patch Correctness Assessment. Recently, researchers often adopt either: (1) manual annotation, i.e., authors of repair techniques manually judge the correctness of patches generated by their and competing tools, or (2) automated assessment, i.e., an independent test suite is used to automatically assess patch correctness. Le et al. [17] showed that manual annotation is more effective but is expensive. Automated assessment, on the other hand, does not require a manual effort but is less effective [17]. Recent research effort has been devoted to automated patch correctness assessment [18], [21], [17]. Existing automated patch correctness assessment techniques assume the oracle (ground truth) patches are available [18], [17], [33]. For example, Xin et al. [18], and Le et al. [17] generate new test cases based on the correct (ground truth) program to identify overfitting patches. Our proposed technique falls into this category wherein we also assume the ground truth patches are available. Different from existing test-based approaches, our technique relies on program invariants to judge patch correctness.

2.2 Program invariants

Program invariants *(invariants for short)* is a term referring to *properties that hold at a certain program point or points, which might be found in an* `assert` *statement, or a formal specification* [22]. For example, a program invariant can be $x >= abs(y)$ or $size(A) == size(B)$. Among several of their usages, program invariants can be used to detect modifications that violate the original properties of a program.

True invariants, however, are usually difficult to obtain in real-world projects, and thus researchers often resort to properties known as likely invariants, which *holds for some executions, but perhaps not all* [34], [35]. Likely invariants can be automatically inferred from execution traces by dynamic invariant detection techniques which generalizes from execution traces using invariant templates. Previous studies have demonstrated the effectiveness of likely invariants in various tasks including complexity analysis [36], [37], termination analysis [38], bug localization [39] or neural network analysis [40], [41].

In this paper, we use Daikon [22] - a popular tool for mining likely invariants, as our dynamic invariant detection technique. Daikon observes the execution traces of programs and matches them against a set of templates to infer likely invariants that hold on all or most of the executions. From a large set of 311 templates (c.f. Details in Daikon Manual Documentation [1]), Daikon can detect a wide variety of invariants that generalize well beyond the test suite used to produce the execution traces [42].

3 MOTIVATION

Let us now use an example to motivate our approach of using program invariants to determine patch correctness. The bug example in Figure 1 shows a APR-generated patch (Figure 1a) and the ground truth developer-written patch (Figure 1b).

In this example, the buggy method is `iterateSimplex()` which computes the next simplex of the multi-directional optimizer. The root cause of the bug is the endless loop `while(true)` (line 6 in Figure

1. http://plse.cs.washington.edu/daikon/download/doc/daikon/Daikon-output.html#Invariant-list

```
1: ---org/apache/commons/math/optimization/direct/MultiDirectional.java
2: +++org/apache/commons/math/optimization/direct/MultiDirectional.java
3: protected void iterateSimplex(final Comparator<ValuePair> comparator)
4: throws OptimizationException
5: {
6: while (true) {
7:    if (++iterations > maxIterations) {
8:        throw new OptimizationException(new
9:               MaxIterationsExceededException(maxIterations));
10:   }
11:                              ...
12:    final RealPointValuePair contracted = evaluateNewSimplex(original,
13:                                                 gamma,comparator);
14:+    if (true)
15:+       return;
16:    if (comparator.compare(contracted, best) < 0) {
17:        return;
18:    }
```

(a) An overfitting patch generated by Kali [27]

```
1: ---org/apache/commons/math/optimization/direct/MultiDirectional.java
2: +++org/apache/commons/math/optimization/direct/MultiDirectional.java
3: protected void iterateSimplex(final Comparator<ValuePair> comparator)
4: throws OptimizationException
5: {
6: while (true) {
7:    if (++iterations > maxIterations) {
8:        throw new OptimizationException(new
9:               MaxIterationsExceededException(maxIterations));
10:   }
11:                              ...
12:    final RealPointValuePair contracted = evaluateNewSimplex(original,
13:                                                 gamma,comparator);
14:    if (comparator.compare(contracted, best) < 0) {
15:        return;
16:    }
17:
19:+    final int iter = getIterations();
20:+    boolean converged = true;
21:+    for (int i = 0; i < simplex.length; ++i) {
22:+        converged &= checker.converged(iter, original[i], simplex[i]);
23:+    }
24:+    if (converged) {
25:+        return;
26:+    }
```

(b) The correct patch written by human developers

Fig. 1: An overfitting patch generated by Kali and the ground-truth human-written patch for Math-84

1b). The developer-written patch as shown in Figure 1b, correctly inserts a code snippet (line 5, lines 19-26 in Figure 1b) to correctly stop the loop once the algorithm converges. Meanwhile, the plausible patch generated by *Kali* [27] in Figure 1a inserts an early `return` at lines 14-15, rendering the failing test case to plausibly pass. The APR-patched program avoids the endless loop, but on the other hand, it ignores the main algorithm, which requires many iterations to converge. Unfortunately, state-of-the-art test-based APAC techniques such as RGT [43], DiffTGen [18] and Randoop [19] failed to identify behavioral differences between the APR-patched program and the ground truth [13] due to the large search space of bug-witnessing test cases.

Invariants come into play. Let us now look at how program invariants can show that the APR-patched program is overfitting. The intended behavior of the program is that under different inputs, the algorithm will terminate accordingly after several iterations once the algorithm converges. For the method `iterateSimplex`, an invariant `iterations > orig(iterations)` is inferred from both the buggy version under repair and the correct version of the program. This invariant means that the value for the variable `iterations` before the method `iterateSimplex` is called, denoted as `orig(...)`, is smaller than that of after the method is executed. The variable `iterations` measures the number of iterations executed by the algorithm. This invariant reflects the fact that the while loop (Line 4 in Figure 1b) should be executed until the multi-directional optimizer converges, which may require many iterations to complete. Meanwhile, in the APR-patched program, the while-loop always terminates after the first iteration due to code snippet `if (true) { return ;}` (line 14-15 in Figure 1a). Indeed, we obtained an invariant `iterations - orig(iterations) - 1 == 0` for the APR-patched program. This invariant means that the value for the variable of `iterations` is always incremented by 1 after the method `iterateSimplex` is executed. This invariant shows an overfitting behavior of the APR-patched program, which violates the intended behaviors of the program, i.e., under varying inputs, the algorithms need varying numbers of iterations to converge. Our technique, INVALIDATOR, detects this overfitting behavior of the APR-patched program by comparing the invariant generated from the APR-patched program versus that of the buggy and ground truth programs. Specifically, INVALIDATOR realizes that the APR-patched program maintains an invariant that never holds in the buggy and ground truth programs, witnessing a behavioral divergence that accounts for a possible error.

4 METHODOLOGY

Figure 2 illustrates the workflow of INVALIDATOR. A APR-patched program is first validated based on a semantic-based classifier. In this phase, INVALIDATOR reason about patch correctness based on specifications, i.e., *correct* and *error specifications*, inferred via analyzing the behaviors differences of buggy program and its correct version, captured using automatically-inferred program invariants. A APR-patched program is considered *overfitting* if invariants inferred from the APR-patched program either violate the correct specification or maintain error specification. In case the inferred specifications fail to determine an overfitting patch, INVALIDATOR further uses a learning-based model, which leverages syntactic reasoning to estimate the probability that the APR-patched program is overfitting. We explain more details of INVALIDATOR below.

4.1 Semantic-based Patch Classifier

Figure 3 describes how INVALIDATOR identifies *overfitting patches* via semantic-based patch classifier. INVALIDATOR first constructs approximate specifications of the program under test via program invariants inferred by a dynamic invariant inference tool, namely DAIKON [22] (Section 4.1.1). Then, based on the inferred specifications, INVALIDATOR automatically classifies whether a APR-patched program is overfitting (Section 4.1.2). Below, we explain each phase of INVALIDATOR in detail.

4.1.1 Invariant-based Specification Inference

The purpose of invariant inference is to ultimately derive specifications that determine the correct and error behaviors of a program. INVALIDATOR uses invariants to approximate the specifications based on two key observations that allow detecting behavioral differences, as follows:

Observation 1. *Program invariants that are maintained in both the buggy and correct (ground truth) versions of a program can serve as correct specification of the original program*

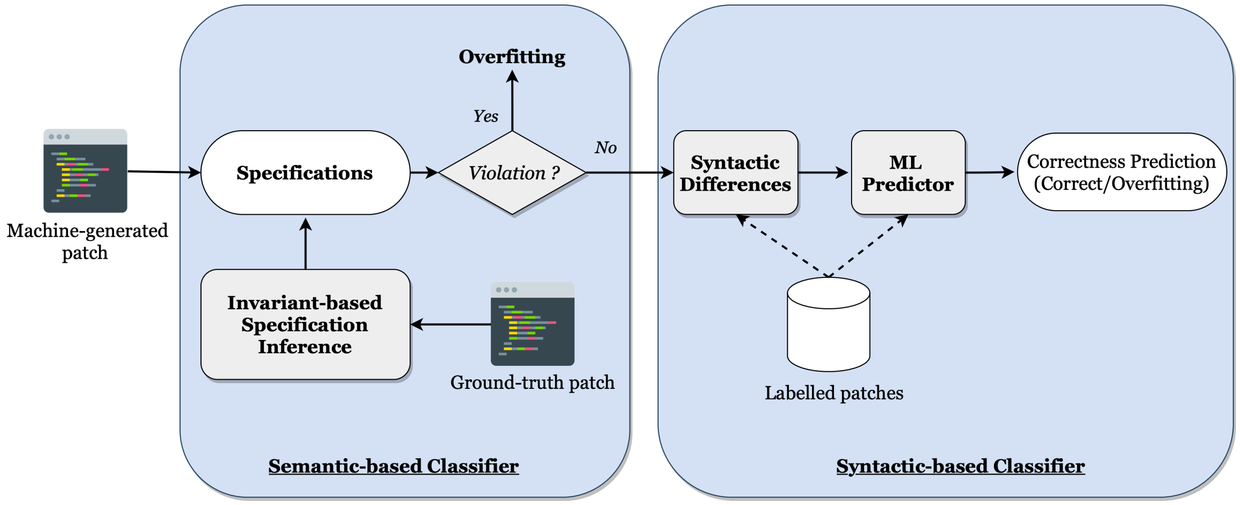

Machine-generated patch
Specifications
Violation ?
Yes
Overfitting
No
Invariant-based Specification Inference
Ground-truth patch
Semantic-based Classifier
Syntactic Differences
ML Predictor
Correctness Prediction (Correct/Overfitting)
Labelled patches
Syntactic-based Classifier

Fig. 2: The workflow of Invalidator

Observation 2. *Program invariants that only exist in the buggy program but do not maintain in the correct version may represent error specification of the buggy program.*

Based on the correct and error specification, INVALIDATOR can heuristically assess the patch correctness. Below, we formally define the correct and error behaviors, explain how we build them via invariant inference, and how they can be used to determine overfitting patches effectively.

We formally describe the correct specifications of a program based on invariants in Definition 1. Correct specifications reflect common behaviors in both the original buggy program and the correct (ground truth) program in which the bug is fixed by developers. We use a set of invariants, denoted as $\mathcal{C}$, which commonly appear in both the buggy version and the correct version of a program, to approximate the correct behaviors of the program. The use of $\mathcal{C}$ reduces the false positive rate (as we shall see in Section 5). Error behaviors as formally defined in Definition 2, on the other hand, capture the behavioral divergence of the buggy program from the correct program. The behavioral difference is reflected by a set of program invariants $\mathcal{E}$ that hold in the buggy program but do not hold in the correct version of the program.

Definition 1. *(Correct specification) Consider a buggy program $\mathbb{B}$ and its correct/ground truth version $\mathbb{G}$. The correct specification of $\mathbb{G}$ is approximated by a set of invariants $\mathcal{C}$ such that $\mathbb{B} \models \mathcal{C}$ and $\mathbb{G} \models \mathcal{C}$.*

Definition 2. *(Error specification) Consider a buggy program $\mathbb{B}$ and its correct/ground truth version $\mathbb{G}$. The error specification of $\mathbb{B}$ is approximated by a set of invariants $\mathcal{E}$ such that $\mathbb{B} \models \mathcal{E}$ and $\mathbb{G} \not\models \mathcal{E}$.*

Let us now explain how we build the specifications that approximate the correct and error behaviors of a program as depicted in Figure 3. Note that we use both the buggy and correct programs to infer the specifications. For each program, denoted as *prog*, INVALIDATOR records executions across both sets of failing test cases $\mathbb{F}$ and set of related passing test cases $\mathbb{P}$. Typically, passing test cases $\mathbb{P}$ reflect correct specification of a program, while failing

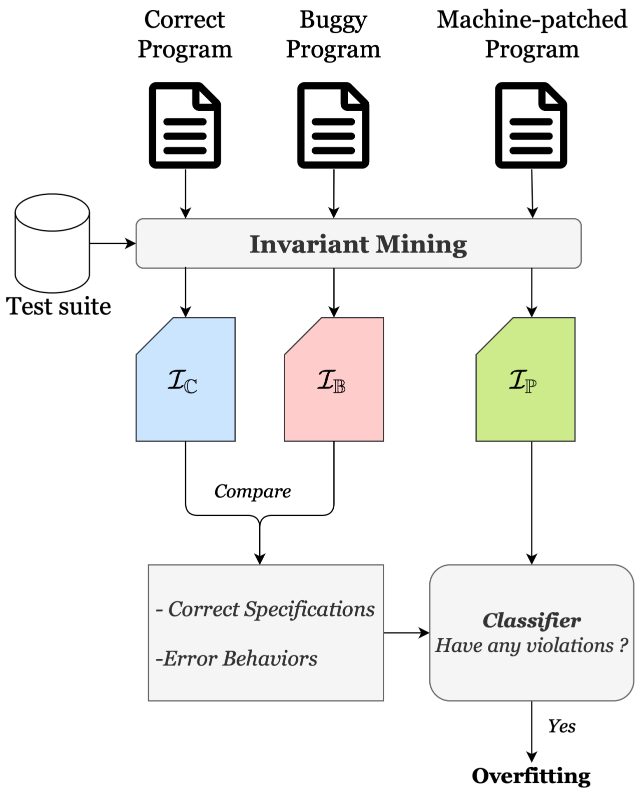


Fig. 3: APAC via invariant-based specification inference. $\mathcal{I}_{\mathbb{C}}$, $\mathcal{I}_{\mathbb{B}}$ and $\mathcal{I}_{\mathbb{P}}$ are sets of invariants inferred from correct program, buggy program and APR-patched program, respectively.

test cases $\mathbb{F}$ reflect error specification of the program. Thus, INVALIDATOR maintains the execution traces of the two test sets $\mathbb{F}$ and $\mathbb{P}$ separately to construct specifications for correct and error specification later on. INVALIDATOR then leverages DAIKON [22] to infer likely invariants based on the execution traces. DAIKON first captures runtime values of variables at specific points of a program, such as the points at which a method is entered or exited, and then it uses a set of templates satisfying the runtime values to infer likely invariants, which are properties that hold over all of the executions. We refer to invariants inferred from passing test cases and failing test cases of a program $prog$ as $\mathcal{I}^F_{prog}$ and $\mathcal{I}^P_{prog}$, respectively.

Let us use $\mathbb{B}$ and $\mathbb{G}$ to denote the original buggy and correct (ground truth) versions of a program. INVALIDATOR then infers invariants from the passing test cases on $\mathbb{B}$ and $\mathbb{G}$, denoted as $\mathcal{I}^P_{\mathbb{B}}$ and $\mathcal{I}^P_{\mathbb{G}}$ respectively. INVALIDATOR constructs the correct specification $\mathcal{C}$ of $\mathbb{G}$ by intersecting the two sets of invariants $\mathcal{I}^P_{\mathbb{B}}$ and $\mathcal{I}^P_{\mathbb{G}}$, taking the resulting invariants as an approximation for correct specification of $\mathbb{G}$. To construct the specifications $\mathcal{E}$ for error specification in $\mathbb{B}$, INVALIDATOR first infers invariants from the failing test cases on $\mathbb{B}$ and $\mathbb{G}$, denoted as $\mathcal{I}^F_{\mathbb{B}}$ and $\mathcal{I}^F_{\mathbb{G}}$ respectively. The specification $\mathcal{E}$ is then approximated by the invariants that are in $\mathcal{I}^F_{\mathbb{B}}$ but are not in $\mathcal{I}^F_{\mathbb{G}}$. In summary, $\mathcal{C}$ represents the expected specification in $\mathbb{B}$ and $\mathbb{G}$, while $\mathcal{E}$ represents the behavioural difference of $\mathbb{B}$ compared to $\mathbb{G}$.

INVALIDATOR uses the constructed specifications $\mathcal{C}$ and $\mathcal{E}$ as input to its patch classifier that we describe in Section 4.1.2 to identify overfitting patches. Note that, INVALIDATOR classifier considers invariants at inferred from all methods executed by a given test suite instead of invariants inferred from buggy methods only (i.e., methods modified by human developers in the correct program) as prior works [44], [26]. We discuss the effectiveness of the classifier with these two granularities in detail in Section 5.

Algorithm 1: Invariant-based Specification Inference. $\mathbb{B}$ is the original buggy program, $\mathbb{P}$ is a APR-patched program, and $\mathbb{G}$ is the correct/ground truth program by developers

Input:
- $\mathcal{I}^P_{\mathbb{P}}$: invariant inferred from passing tests on $\mathbb{P}$
- $\mathcal{I}^F_{\mathbb{P}}$: invariant inferred from failing tests on $\mathbb{P}$
- $\mathcal{I}^P_{\mathbb{B}}$: invariant inferred from passing tests on $\mathbb{B}$
- $\mathcal{I}^F_{\mathbb{B}}$: invariant inferred from failing tests on $\mathbb{B}$
- $\mathcal{I}^P_{\mathbb{G}}$: invariant inferred from passing tests on $\mathbb{G}$
- $\mathcal{I}^F_{\mathbb{G}}$: invariant inferred from failing tests on $\mathbb{G}$

Output: True: $\mathbb{P}$ is overfitting, False: Otherwise

```
1  $\mathcal{C} \leftarrow \mathcal{I}^P_{\mathbb{G}} \cap \mathcal{I}^P_{\mathbb{B}}$          ▷ Correct specification
2  $\mathcal{E} \leftarrow \mathcal{I}^F_{\mathbb{B}} \setminus \mathcal{I}^F_{\mathbb{G}}$          ▷ Error specification
3  foreach $inv$ in $\mathcal{C}$ do
4      if $inv \notin \mathcal{I}^P_{\mathbb{P}}$ then
5          return True
6      end
7  end
8  foreach $inv$ in $\mathcal{E}$ do
9      if $inv \in \mathcal{I}^F_{\mathbb{P}}$ then
10         return True
11     end
12 end
13 return False
```

4.1.2 Patch Classifier

The patch classifier takes as input a APR-generated patch, the constructed specifications including correct specification $\mathcal{C}$ and error specification $\mathcal{E}$ to determine whether the patch is overfitting.

Our approach to identifying patch correctness hinges on the following two key observations:

Observation 3. *A patch should be considered overfitting if it violates any of the correct specification described in $\mathcal{C}$.*

Observation 4. *A patch should be considered overfitting if it maintains any of the error specification described in $\mathcal{E}$.*

The above observations translate to the following two rules that allow INVALIDATOR to determine whether a APR-patched program is overfitting. Let $\mathbb{B}$ be a buggy program, $\mathbb{G}$ be the human-written correct version of the program, $\mathbb{P}$ be a APR-patched program to be assessed whether it is overfitting, and $\mathcal{I}_{\mathbb{P}}$ be the set of invariants inferred from $\mathbb{P}$. Then, a patch is considered as overfitting if it satisfies either of the following:

- **Overfitting-1**: The patch violates the specifications representing correct specification $\mathcal{C}$ for $\mathbb{B}$ and $\mathbb{G}$. More formally, $\exists inv \in \mathcal{C} : inv \notin \mathcal{I}_{\mathbb{P}}$
- **Overfitting-2**: The patch maintains any error behaviors described in $\mathcal{E}$ for $\mathbb{B}$. More formally, $\exists inv \in \mathcal{E} : inv \in \mathcal{I}_{\mathbb{P}}$

In the *Overfitting-1* rule, we consider any APR-patched program $\mathbb{P}$ to violate the correct specification if the set of invariants inferred from $\mathbb{P}$, denoted as $\mathcal{I}_M$, excludes any invariants that are in the correct specifications $\mathcal{C}$. This helps guard against cases where the patch deletes some functionalities of the original program and thus excludes the specifications corresponding to the functionalities. In the *Overfitting-2* rule, any patch that still maintains an invariant representing error specification in the original buggy program $\mathbb{B}$ is considered overfitting.

Note that, INVALIDATOR needs to compare an invariant to another to determine whether a patch falls into either of the overfitting rules we defined above. INVALIDATOR achieves this by comparing invariants syntactically and semantically. If two invariants are not syntactically the same, INVALIDATOR leverages a SMT solver, i.e., Z3 [45] to determine if they are semantically equivalent. Generally, two logical formulae $\mathcal{A}$ and $\mathcal{B}$ are equivalent if $(\mathcal{A} \Rightarrow \mathcal{B}) \wedge (\mathcal{B} \Rightarrow \mathcal{A})$. For example, $a >= b$ and $b <= a$ are syntactically different but are determined as semantically equivalent by Z3; Z3 determines that the formulae $(a >= b \Rightarrow b <= a) \wedge (b <= a \Rightarrow a >= b)$ is satisfiable and hence $a >= b$ is equivalent to $b <= a$.

4.1.3 Optimization via test selection

Our observation is that not all test cases are relevant to the bug at hand. Before running DAIKON, Invalidator performs *test selection* described in Algorithm 2 to select a subset of passing test cases that are related to the bug to collect execution traces. This way, the reduced test suite helps Invalidator optimizes for the running time by using DAIKON without compromising the accuracy of Invalidator.

4.2 Syntactic-based Patch classifier

In case INVALIDATOR fails to reason about patch correctness via invariant-based specification inference (described

Algorithm 2: Test selection

```
Input:
    • ℙ: program
    • ℙ: set of modified methods
    • 𝕋: set of test cases
Output: Related tests
1 $\mathbb{R} \leftarrow \emptyset$                              ▷ Set of related tests
2 foreach $test$ in $\mathbb{T}$ do
3     $t \leftarrow Coverage(\mathbb{P}, test)$    ▷ Test coverage if $t$ cover
      at least one method $\in \mathbb{P}$ then
4         $\mathbb{R} \leftarrow \mathbb{R} \cup \{test\}$
5     end
6 end
7 return $\mathbb{R}$
```

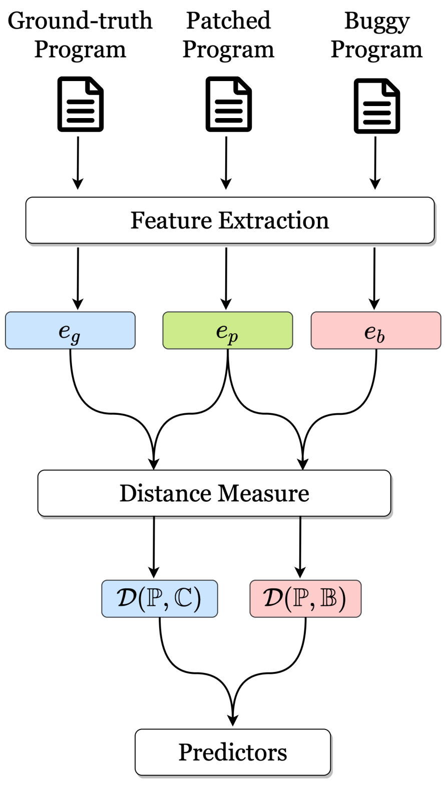


Fig. 4: Model architecture of the syntactic classifier. e_b, e_p, e_c are representation of buggy program, patched program and groundtruth program, respectively. $\mathcal{D}(\mathbb{P}, \mathbb{B})$, $\mathcal{D}(\mathbb{P}, \mathcal{G})$ are distances from patched program to buggy program and correct program

in Section 4.1), INVALIDATOR resorts to estimating the probability that a APR-patched program being overfitting by measuring syntactic proximity of the patch to the buggy and ground truth programs. Given a APR-patched program $\mathbb{P}$, INVALIDATOR first measures the syntactic differences between $\mathbb{P}$ and the buggy program $\mathbb{B}$, denoted as $\mathcal{D}(\mathbb{P}, \mathbb{B})$, and between $\mathbb{P}$ and its ground-truth program $\mathbb{G}$, denoted as $\mathcal{D}(\mathbb{P}, \mathbb{G})$. INVALIDATOR employs a pretrained language model (i.e., CODEBERT [46]) to extract syntactic features of these programs and then uses comparison functions [47] as distance measures to identify syntactic differences between them. Finally, INVALIDATOR uses a machine learning model to predict patch correctness. Figure 4 illustrates the classification pipeline of our syntactic-based classifier. Below, we explain each component of the pipeline in detail.

4.2.1 *Feature Extraction*

The feature extraction layers aim to extract the embedding vector (a.k.a. features) representing syntactic information of buggy, patched, and correct programs. To do this, we utilize CODEBERT [46], a powerful pre-trained model for general-purpose representations of source code that have been demonstrated its effectiveness on various software engineering tasks [46], [48], [49], [50]. CODEBERT takes a code fragment as its input, then uses a tokenizer (i.e., Roberta tokenizer) to tokenize the given code to the sequence of tokens. Finally, it forwards the sequence to a pre-trained multi-layer bidirectional Transformer [51] to obtain a corresponding numerical vector. More specifically, given a code fragment, INVALIDATOR employs CODEBERT to represent the code fragment as the vector defined as follows:

$$e_{code} = \langle v_1, v_2, \dots, v_k \rangle \tag{1}$$

where $k = 768$ is the embedding dimension of CODEBERT. For convenience, we denote e_b, e_p, e_c as representation of buggy program, patched program and correct program, respectively.

4.2.2 *Distance Measure*

The goal of the distance measure layers is to build the vectors that capture the syntactic differences between APR-patched program and buggy program and correct program. Inspired by prior works [24], [52] on program repair, we leverage comparison functions [47] to represent various types of syntactic differences. The distance measure layers take as input the embedding vectors of buggy program, patched program, and correct program (denoted by e_b, e_p, e_c, respectively) and output the vectors representing the syntactic difference of APR-patched program compared to buggy program and correct program. These vectors are then concatenated to represent distance vectors, which have $2 \times k + 2$ dimensions where k is the dimension of code embeddings (i.e., 768). In this paper, we use three comparison functions, consisting of Cosine similarity, Euclidean distance, and element-wise subtraction. We briefly explain these comparison functions below.

Element-wise subtraction. We perform element-wise subtraction for the embedding vectors of the APR-patched program and the buggy program and the correct program as follows:

$$e_1^{\text{sub}} = e_p - e_b$$

$$e_2^{\text{sub}} = e_p - e_c$$

Element-wise multiplication. We perform element-wise multiplication for the embedding vectors of the APR-patched program and the buggy program and the correct program as follows:

$$e_1^{\text{mul}} = e_p \odot e_b$$

$$e_2^{\text{mul}} = e_p \odot e_c$$

where $\odot$ is the element-wise multiplication operator.

Euclidean Distance. We capture the distance between the embedding vectors of the APR-patched program and the buggy program and the correct program based on Euclidean distance as follows:

$$e_1^{\text{euc}} = \|e_p - e_b\|$$

$$e_2^{\text{euc}} = \|e_p - e_c\|$$

where $\|\cdot\|$ is the Frobenius norm.

Cosine Similarity. We capture the similarity between the embedding vectors of the APR-patched program and the buggy program and the correct program based on Cosine similarity as follows:

$$e_1^{\text{sim}} = \frac{e_p e_b}{\|e_p\| \, \|e_b\|}$$

$$e_2^{\text{sim}} = \frac{e_p e_c}{\|e_p\| \, \|e_c\|}$$

where $\|\cdot\|$ is the Frobenius norm.

Distance vector. Finally, we concatenated the vectors resulting from applying these three different comparison functions to represent the syntactic distances from patched program to buggy program and correct program as follows:

$$\mathcal{D}(\mathbb{P}, \mathbb{B}) = \mathbf{e}_1^{\text{sub}} \oplus \mathbf{e}_1^{\text{mul}} \oplus \mathbf{e}_1^{\text{euc}} \oplus \mathbf{e}_1^{\text{sim}}$$

$$\mathcal{D}(\mathbb{P}, \mathbb{C}) = \mathbf{e}_2^{\text{sub}} \oplus \mathbf{e}_2^{\text{mul}} \oplus \mathbf{e}_2^{\text{euc}} \oplus \mathbf{e}_2^{\text{sim}}$$

where $\oplus$ is concatenation operation, $\mathcal{D}(\mathbb{P}, \mathbb{B})$ and $\mathcal{D}(\mathbb{P}, \mathbb{G})$ are distances from patched program to buggy program and correct program.

4.2.3 Predictor

Given the above distance vectors, we leverage a machine learning model to predict patch correctness from labeled data. Following the finding of Tian et al. [24] that Logistic Regression applied to BERT embeddings yields the best performance in predicting patch correctness, we consider the Logistic Regression algorithm as our predictor. Logistic regression is a well-known machine learning (ML) algorithm that predicts patch correctness based on a linear transform and logistic loss function.

4.2.4 Correctness Prediction

INVALIDATOR classifies a patch as correct or overfitting based on prediction score, i.e., probability that a given patch is overfitting, produced by ML-based predictor. Let $\mathcal{P}(m)$ denotes the prediction score of a APR-generated patch m. We determine the correctness of a given patch m using the following formula:

$$\text{correctness } (m) = \begin{cases} \text{correct} & \mathcal{P}(m) \leq \mathcal{T} \\ \text{overfitting} & \mathcal{P}(m) > \mathcal{T} \end{cases}$$

where $\mathcal{T}$ is our classification threshold.

TABLE 1: Dataset for evaluating automated patch correctness assessment techniques

Dataset	Correct patches	Overfitting patches	Total
Xiong et al. [21]	30	109	139
Wang et al. [44]	216	450	666
DEFECTS4J [53]	223	0	223
Final dataset	**377**	**508**	**885**

TABLE 2: The statistics of evaluation and training dataset

Dataset	Correct patches	Overfitting patches	Total
Training	331	340	671
Validation	16	59	75
Evaluation	30	109	139

5 EMPIRICAL EVALUATION

In this section, we empirically evaluate Invalidator on a dataset of patches generated by well-known automated program repair techniques for bugs in large real-world Java programs. We discuss the dataset, experimental settings, and metrics in Section 5.1. Section 5.2 lists our research questions, followed by our findings in Section 5.3.

5.1 Experimental Settings

5.1.1 Dataset

To evaluate the effectiveness of automated patch correctness assessment (APCA) techniques, we have collected a data set of APR-generated patches whose correctness labels are manually identified by independent developers and researchers. Particularly, we used 220 patches released by Xiong et al. [21] and 902 patches released by Wang et al. [44]. Following previous works [18], [24], we only consider patches from four widely-used projects in DEFECTS4J: Chart, Time, Lang, and Math. As a result, we have 139 patches and 666 patches from Xiong et al.'s and Wang et al.'s dataset, respectively. Next, to reduce data imbalance issues, i.e., very few APR-generated patches are actually labeled as correct, we also supplement the dataset with developer-written patches as supplied in the DEFECTS4J dataset following [24]. This results in 1028 patches including 469 correct patches and 559 overfitting patches.

Following previous work [24], [21], [24], [23], we consider 666 patches from Wang et al.'s dataset and 223 developer's patches from DEFECTS4J [53] as the training and validation set and 139 patches from Xiong et al. [21] as evaluation set. As there may be a duplication between Wang et al. 's dataset, Defects4J's patches and Xiong et al.'s dataset, we removed the duplicated patches to avoid data leakage. Particularly, we removed a patch from the training and validation set if it is syntactically equivalent to a patch in the evaluation set. As a result, we obtained 746 (out of 889) patches, including 347 correct patches 559 and overfitting patches, for the training and validation phase. From the 746 patches, we use 90% of patches (671 patches) for training our learning model, and the remaining 75 patches are used for validation. Table 1 and Table 2 shows the details of patches considered in our experiments and the characteristics of our training, validation and evaluation datasets respectively.

5.1.2 Evaluation Metrics

By using the data set described in Section 5.1.1, we assess the effectiveness of automated patch correctness assessment (APCA) techniques by comparing the labels produced by APCA versus the ground truth labels. Furthermore, we aim to assess how many patches an APCA technique produces that match with that of the ground truth labels. Specifically, we use standard metrics of classification problems [54], [55], *Recall* (Equation 2), *Precision* (Equation 3), *Accuracy* (Equation 4) and *F-Measure* (Equation 5); they are defined by the following metrics:

- **True Positive (TP)**: a generated patch is labeled as "overfitting" by both an APCA technique and the ground truth.
- **False Positive (FP)**: a generated patch is labeled as "overfitting" by an APCA technique but is labeled as "correct" by the ground truth.
- **True Negative (TN)**: a generated patch is labeled as "correct" by both an APCA technique and the ground truth.
- **False Negative (FN)**: a generated patch is labelled as "correct" by an APCA technique, but is labelled as "overfitting" by the ground truth.

$$\mathbf{Recall} = \frac{TP}{TP + FN} \tag{2}$$

$$\mathbf{Precision} = \frac{TP}{TP + FP} \tag{3}$$

$$\mathbf{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{4}$$

$$\mathbf{F-Measure} = \frac{2 \text{ x } Recall \text{ x } Precision}{(Precision + Recall)} \tag{5}$$

Among these evaluation measures, *Recall* verifies whether an approach can successfully classify overfitting patches. A higher *Recall* is demanded by developers as we do not want to waste their efforts on analyzing a substantial number of overfitting patches [15]. Meanwhile, *Precision* measures the proportion of discarded patches by an approach that is genuinely overfitting. A higher *Precision* is desired by program repair research as we do not want to discard correct patches [23].

However, the comparison of APAC techniques that relies only on *Recall* or *Precision* may be incomplete. For example, a APAC technique can only consider patches as overfitting if it violates strict conditions (e.g., a high confidence value) to achieve a higher *Precision*, which could result in a low *Recall*. On the contrary, an approach can classify all patches as overfitting to achieve perfect *Recall*, which results in low Precision. To address these issues, we consider *F-Measure* and *Accuracy* as additional evaluation metrics to measure the performance of APAC techniques, *F-Measure* seeks a balance between *Recall* and *Precision* while *Accuracy* is the comprehensive evaluation of all TP, FP, TN, and FN.

Besides, we also consider Area Under the Curve (denoted as **AUC**), which is defined as follows.

$$\mathbf{AUC} = \frac{S_0 - n_0(n_0+1)/2}{n_0 n_1} \tag{6}$$

where n_0 and n_1 are the numbers of overfitting and correct patches, respectively, and $S_0 = \Sigma r_i$, where r_i is the rank of the i^{th} overfitting patch in the descending list of output probability produced by each model.

AUC is a widely-used metric to evaluate the effectiveness of threshold-dependent classifiers [56], [57]. In our paper, *AUC* is essential to compare the performance of syntactic-based classifiers.

5.1.3 Implementation Details

For INVALIDATOR, we implement the proposed approach using Python programming language. For CODEBERT model, we use HuggingFace's Transformers framework [2] as recommended by their authors in CODEBERT's github repository [3]. With respect to the threshold $\mathcal{T}$ of syntactic-based classifier, we set the default threshold at 0.975. To choose a classification threshold, we constraint the threshold to avoid filtering out any correct patches as following prior works [21], [24]. Note that, we tune our classification threshold on an independent validation set (see details in Section 5.1.1) instead of evaluation set as prior works [21], [24] to avoid overfitting. We also investigate the impact of the threshold on the performance of INVALIDATOR in Section 5.

With respect to baseline techniques, we collect results of ODS, PATCHSIM, ANTI-PATTERNS, and BERT + LR from prior works [23], [21], [24]. For DIFFTGEN and GT-INVARIANT, we run their implementation to obtain their prediction for Xiong et al. dataset due to the lack of the result in the literature.

5.2 Research Questions

Our evaluation aims to answer these research questions:

RQ1: *How effective is our approach to validate patches generated by automatic repair tools?*
The research question concerns the ability of INVALIDATOR for identifying overfitting patches generated by automated program repair techniques. To demonstrate the value of our approach for automated patch correctness assessment tasks, we conduct an experiment in a dataset of 885 APR-generated patches (as described in Section 5.1.1) in terms of *Precision*, *Recall*, *Accuracy* and *F-Measure*. Then, we compare our approach to state-of-the-art baseline techniques, namely:

- RGT: Ye et al. [43] proposed to use a testing procedure, named Random Testing with Groundtruth (RGT) [58], for APAC. Particularly, RGT automatically generates tests based on developer-patched programs, which encodes the correct program behavior. And then if any automatically generated test fails on a APR-patched program, RGT considers the program as *overfitting*.
- ODS: Ye et al. [23] proposed ODS, an overfitting detection system. ODS builds machine learning classifiers based on 4,199 manually-crafted features for classifying overfitting patches;
- BERT + LR: Tian et al. [24] proposed learning-based APAC technique that utilize BERT [59] and Logistic

2. https://huggingface.co/docs/transformers/index
3. https://github.com/microsoft/CodeBERT

Regression to learn representations of code changes from historical data to predict the correctness of APR-generated patches. In this paper, we refer to their technique as BERT + LR;
- PATCHSIM: Xiong et al. [21] proposed a dynamic APAC technique based on the similarity of execution trace similarity. In this paper, we refer to their technique as PATCHSIM;
- DIFFTGEN: Xin et al. [18] proposed a APAC technique that identifies overfitting patches through test case generation. DIFFTGEN is the closest baseline related to our approach. Both INVALIDATOR and DIFFTGEN assume the ground truth patches are available;
- ANTI-PATTERNS: In [25], the authors proposed seven generic categories of program transformation to detect overfitting patches. In this paper, we refer to their technique as ANTI-PATTERNS;
- GT-INVARIANT: Recently, Yang and Yang [26] found that the majority of the overfitting patches expose different runtime behaviors captured by GT-INVARIANT's invariant [22]. Based on their findings, Wang et al. [44] adopt a simple heuristic that considers a machine generated-patch is *overfitting* if there is inferred invariant of this patch is different from that of the correct program. In this paper, we refer to their technique as GT-INVARIANT;

RQ2: *How effective is our syntactic-based classifier?*

This research question investigates the effectiveness of our syntactic-based classifier in assessing patch correctness. Toward this, we conduct experiments to answer two sub-questions:

- **RQ2.1:** *How does our syntactic-based classifier compare to existing techniques?* In this research question, we compared our syntactic-based classifier to existing techniques, including ODS and BERT+LR in terms of *Precision*, *Recall Accuracy*, and *F-Measure* as RQ1. Besides, we also compare the performance of these techniques on *AUC*, a widely-used metric to evaluate the effectiveness of threshold-dependent classifiers.
- **RQ2.2:** *How do our syntactic features compare to existing features?* In this research question, we investigate the effectiveness of our syntactic features extracted from CODEBERT, compared to syntactic features extracted from existing methods, i.e., ODS and BERT.

RQ3: *How does the classification threshold affect the overall performance?*
INVALIDATOR uses a classification threshold to decide whether a patch is overfitting based on the prediction score of Machine Learning-based predictors. For the first two research questions, we set the classification threshold at 0.975, which produces the highest Precision on the validation dataset for Invalidator. In this research question, we investigate the impact of threshold sensitivity on the performance of INVALIDATOR. To do this, we perform experiments towards providing answers for two sub-questions:

- **RQ3.1:** *How does the classification threshold affect the overall performance?* We systematically set different values for this threshold and investigate how this threshold affects the result of INVALIDATOR
- **RQ 3.2:** *How does threshold sensitivity affect our approach compared to other techniques?* The research question aims to investigate the impact of threshold sensitivity on the performance of INVALIDATOR compared to existing threshold-dependent techniques such as PATCHSIM or ODS.

RQ4: *Which components of* INVALIDATOR *contribute to its performance?*

This research question analyzes the contribution of different INVALIDATOR's components to the overall performance. We first investigate the contribution of semantic and syntactic classifiers to the overall performance of INVALIDATOR. Then, we investigate the impact of the design choice of each component, i.e., the granularity of invariants and overfitting rules, on the performance of INVALIDATOR. Specifically, we perform experiments to answer three sub-questions as follows:

- **RQ4.1:** *How does semantic and syntactic classifier affect the performance of our approach?* INVALIDATOR contains two main components, i.e., semantic classifier and syntactic classifier. In this research question, we investigate the contribution of each classifier in an ablation study by dropping each classifier and observing the change in INVALIDATOR's performance.
- **RQ4.2:** *Do invariants inferred from executed methods boost the performance of Invalidator compared to invariants inferred from buggy methods only?* By default, INVALIDATOR classifier considers invariants inferred from all methods executed by a given test suite instead of invariants inferred from buggy methods only (i.e., methods modified by human developers in the correct program) as prior works [44], [26]. In this research question, we investigate the effectiveness of INVALIDATOR with these two granularities.
- **RQ4.3:** *How do overfitting rules affect the performance of our semantic classifier?* By default, our semantic classifier uses a combination of both the *Overfitting-1* and *Overfitting-2* rules described in Section 4.1.2 for identifying overfitting patches. In this research question, we individually compare these two overfitting rules to evaluate their contribution to INVALIDATOR's effectiveness.

5.3 Findings

5.3.1 *RQ1: Effectiveness*

We report the comparison of our approach, INVALIDATOR against baseline techniques consisting of RGT [43], ODS [23], BERT+LR [24], PATCHSIM [21], DIFFTGEN [18], ANTI-PATTERNS [25], GT-INVARIANT [26] on 139 APR-generated patches collected by Xiong et al. [21]. Table 3 presents the detailed results with respect to evaluation metrics given in Section 5.1.2, including: *Recall*, *Precision*, *Accuracy* and *F-Measure*. We highlight the best result for each evaluation metrics as bold numbers. The bold red number denotes the metrics for which the Invalidator shows the highest results among the techniques. Overall, INVALIDATOR successfully identifies correctly 86 out of 109 overfitting patches and misclassified 3 out of 30 correct patches,

TABLE 3: Comparison of the effectiveness of Invalidator with the state-of-the-art techniques. The bold number denotes the best result for Accuracy and F1.

Techniques	TP	FN	FP	TN	Recall	Precision	Accuracy	F1
ANTI-PATTERN	27	82	1	29	0,25	0,96	0,40	0,39
DIFFTGEN	16	93	0	30	0,15	1,00	0,33	0,26
PATCHSIM	62	47	0	30	0,57	1,00	0,66	0,73
GT-Invariant	59	50	7	23	0,54	0,89	0,59	0,67
BERT + LR	43	66	0	30	0,39	1,00	0,53	0,57
ODS	70	39	5	25	0,64	0,93	0,68	0,76
RGT	70	39	5	25	0,64	0,93	0,68	0,76
Invalidator	86	23	3	27	0,79	0,97	**0,81**	**0,87**

equivalent to scores of 0.79, 0.97, 0.81 and 0.87 in terms of *Recall*, *Precision*, *Accuracy* and *F-Measure*, respectively. This implies that Invalidator outperforms all baselines in *Recall*, *Accuracy* and *F-Measure* and obtains a good *Precision* of 0.97. We present more details below.

Accuracy. Table 3 shows that INVALIDATOR successfully identifies correctly 86 out of 109 overfitting patches and 27 out of 30 correct patches, equivalent to *Accuracy* of 0.81. This implies that Invalidator outperforms the best baselines (i.e., ODS and RGT) by 19 % and shows improvements of 23% - 146% compared to the other baselines.

F-Measure INVALIDATOR outperforms the two best baselines (i.e. ODS and RGT) by 14 %. In detail, INVALIDATOR outperforms BERT+LR, PATCHSIM, DIFFTGEN, ANTI-PATTERNS, GT-INVARIANT by 54%, 20%, 239%, 120%, 29%, respectively. This is mainly because INVALIDATOR successfully identifies 79% of overfitting patches while the best baselines only filter out 64% of overfitting patches while still keeping an acceptable precision of 0.97.

Recall. In terms of *Recall*, INVALIDATOR achieves the improvements of 23% (0.79 vs. 0.64) compared to the best baselines (i.e., ODS and RGT). More specifically, INVALIDATOR outperforms RGT, ODS, BERT+LR, PATCHSIM, DIFFTGEN, ANTI-PATTERNS, GT-INVARIANT by 23%, 23%, 100%, 39%, 438%, 219%, 46%, respectively. This is mainly because INVALIDATOR utilizes both syntactic and semantic reasoning while other techniques only consider semantic or syntax alone.

Precision. In terms of *Precision*, INVALIDATOR achieves the scores of 0.97, performing better than ANTI-PATTERNS, ODS, RGT and GT-INVARIANT, which show *Precision* of 0.96, 0.93, 0.93 and 0.89, respectively. With respect to BERT+LR, PATCHSIM and DIFFTGEN, our approach slightly underperforms these techniques in terms of *Precision*. However, BERT+LR and PATCHSIM avoid filtering out correct patches by directly tuning the threshold of their classifier on the evaluation set, raising concerns of overfitting on the set. Different from these techniques, we tune our classification threshold on an independent validation set (as presented in Section 5.1.3) to avoid overfitting, leading to lower performance of *Precision* than BERT+LR and PATCHSIM on the evaluation set. Meanwhile, DIFFTGEN, although having a perfect *Precision* (i.e., 1.0), is much less effective in filtering out overfitting patches reflected by a low *Recall* of 0.25.

Complementarity with ODS and RGT. We also perform a detailed analysis on the overfitting patches correctly classified by INVALIDATOR, ODS, and RGT. Figure 5 shows the intersection of their correctly classified overfitting patches. We can see that these techniques only detected 31/109 overfitting patches together, accounting for less than 40% of overfitting patches correctly classified by each technique. Meanwhile, INVALIDATOR, RGT, and ODS individually detect 10, 7, and 5 overfitting patches that are not detected by one another, respectively. More interestingly, the overfitting patches correctly classified by the three techniques cover most of the overfitting patches (107/109). These results suggest that the three techniques are complementary and can be used together to obtain a better patch correctness assessment.

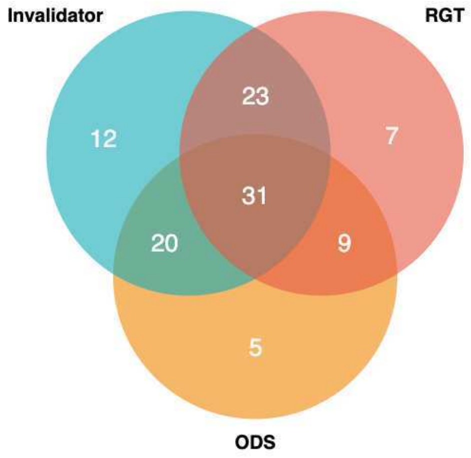


Fig. 5: Intersection on the correctly classifier overfitting patches by INVALIDATOR, ODS and RGT

Case study of unique overfitting patches. To further provide insights about our approach, we manually analyze unique overfitting patches that are possibly detected with the help of the novel techniques in INVALIDATOR. In Figure 6, we give an example of an overfitting patch generated for a bug Math-58, which is detected as overfitting by INVALIDATOR but not RGT and ODS. In this bug, method `fit()` (line 3 in Figure 6b) is used to fit a Gaussian function to the observed points. Ideally, the method must catch the exceptions of observed points having a negative standard deviation and return NaN values. To do this, the method must call method `fit2()` to initialize a new Gaussian function and catch the exceptions before calling method `fit3()` to fit the Gaussian function. In the buggy version, `fit()` directly initializes a new Gaussian function and calls `fit3()` (line

```
1:  ---org/apache/commons/math/analysis/function/Gaussian.java
2:  +++org/apache/commons/math/analysis/function/Gaussian.java
3:    if (param.length != 3) {
4:       throw new DimensionMismatchException(param.length, 3);
5:    }
6: + if ((param[2]) == 0) {
7:       if (param[2] <= 0) {
8:           throw new NotStrictlyPositiveException(param[2]);
9:       }
10:   }
11:   }
12:+ }
```

(a) An overfitting patch generated by Nopol [4]

```
1:  ---org/apache/commons/math/optimization/fitting/GaussianFitter.java
2:  +++org/apache/commons/math/optimization/fitting/GaussianFitter.java
3:    public double[] fit() {
4:       final double[] guess = (new ParameterGuesser(getObservations())).guess();
5: -     return fit3(new Gaussian.Parametric(), guess);
6: +     return fit2(guess);
7:    }
```

(b) The correct patch written by human developers

Fig. 6: An overfitting patch generated by Nopol and the human-written patch for Math-58

5 in Figure 6b). Consequently, the buggy version misses the observed points having a negative standard deviation and throws `NotStrictlyPositiveException`. As we can see in Figure 6a, Nopol fixes the bug by adding the condition `param[2] == 0` (line 6). In this way, if the observed points have a negative standard deviation, i.e., `param[2] < 0`, the `NotStrictlyPositiveException` (line 8) is unreachable. Consequently, the failing test case is plausibly passed, but the program is still incorrect. However, as it is no longer possible to trigger this error, RGT, which relies on test case generation, fails to detect the different behaviors between the overfitting patch and the correct patch. However, INVALIDATOR, which relies on program invariants, still can correctly detect the overfitting patch. Indeed, in both buggy program and Nopol's patched program, INVALIDATOR found the invariant `f.getClass() == Gaussian$Parametric.class` at the entry point of method `fit3()`, indicating that the Gaussian function is directly initialized in method `fit()`. Meanwhile, the Gaussian function should be initialized in `fit2()` reflected by an invariant of the developer-patched program `f.getClass() == GaussianFitter$1.class` at the entry point of method `fit3()`. We can see that Nopol's patch satisfies our Overfitting-2 rule, i.e., maintaining error behavior. Therefore, INVALIDATOR can correctly classify the patch as overfitting. Another example can be seen in Section 3, in which an overfitting patch generated by Kali [27] cannot be detected by RGT but by INVALIDATOR as the patch satisfies our Overfitting-1 rule, i.e., violating correct behavior.

Answers to RQ1: INVALIDATOR yields very promising performance on assessing the correctness of APR-generated patches (*Accuracy* at 0.81 and *F-Measure* at 0.87) and outperforms the best baseline by 19% and 14% in terms of *Accuracy* and *F-Measure*, respectively. Beside, the complementary use of three best performing techique can cover 107/109 overfitting patches.

5.3.2 RQ2: Effectiveness of syntactic-based classifier

[RQ2.1: Our syntactic-based classifier vs. existing techniques]

In this sub-question, we compare the performance of our syntactic-based classifier with two existing learning-based APAC techniques: ODS and BERT+LR. Table 4 presents the effectiveness of our approach and two baselines on six evaluation metrics including *Accuracy*, *F1* and *AUC*. The experimental results demonstrate that INVALIDATOR significantly outperforms two baseline over six evaluation metrics. Particularly, INVALIDATOR yields an *Accuracy* of 0.73 and *F1* of 0.80, outperforming the best baseline, i.e., ODS, by 6% and 5%, respectively. Note that ODS requires manual efforts to extract hand-crafted features while our patch classifier automatically extracts features based on labeled datasets. Compared to BERT+LR, which also uses automatically-extracted features, our syntactic-classifier shows substantial improvement of 38% and 41% in terms of *Accuracy* and *F1*, respectively. Moreover, INVALIDATOR also improves ODS and BERT+LR by 6% and 16% over AUC, indicating that our syntactic classifier has a better discriminative capability than existing techniques regardless of thresholds.

Answers to RQ2.1: Our syntactic-based classifier significantly outperforms existing techniques over all evaluation metrics. Notably, our classifier also improves the best baseline by 6% in terms of AUC, indicating it is more effective than existing techniques regardless of thresholds.

[RQ2.2: Our syntactic features vs. existing syntactic features]

In this sub-question, we compare the performance of our syntactic features extracted from CodeBERT with existing ones extracted from ODS and BERT regarding our syntactic-based classifier. To ease our presentation, we refer to the features as CodeBERT's, ODS's and BERT's features, respectively. Table 5 presents the effectiveness of six variants of the syntactic-based classifier using three syntactic features: ODS's, BERT's, and CodeBERT's features with and without ground-truth knowledge. The evaluation results showed that CODEBERT's features significantly outperform ODS's and BERT's features. Particularly, with ground-truth knowledge, BERT's features show an improvement of 9%, 8%, and 7% regarding *Accuracy*, *F1*, and *AUC*, respectively. Meanwhile, the improvements without ground-truth knowledge are 5%, 7%, 17%. Besides, we also can see that our classifier with ground-truth knowledge improves the variants without the knowledge regardless of syntactic features over three metrics: *Accuracy*, *F1*, and *AUC*. The improvement is especially substantial regarding threshold-dependent techniques, i.e., *Accuracy* and *F1*. These results indicate the advantage of adding ground truth knowledge for syntactic-based classifiers.

Answers to RQ2.2: CodeBERT's features are the most suitable features for our syntactic-based classifier. Besides, ground truth knowledge is helpful for syntactic-based classifiers.

TABLE 4: Comparison of the effectiveness of INVALIDATOR's syntactic-based classifier with the state-of-the-art techniques. The bold number denotes the best result for Accuracy, F1 and AUC.

Techniques	TP	FN	FP	TN	Recall	Precision	Accuracy	F1	AUC
BERT+LR	43	66	0	30	0.39	1.00	0.53	0.57	0.77
ODS	70	39	5	25	0.64	0.93	0.68	0.73	0.84
INVALIDATOR$_{Syn}$	74	35	3	27	0.68	0.96	**0.73**	**0.80**	**0.89**

TABLE 5: Comparison of the effectiveness of CODEBERT features with ODS and BERT features. The bold number denotes the best result for Accuracy. F1 and AUC.

Ground-truth	Techniques	TP	FN	FP	TN	Recall	Precision	Accuracy	F1	AUC
No	**BERT$_{wo-gt}$**	33	76	2	28	0.30	0.94	0.44	0.46	0.71
	ODS$_{wo-gt}$	25	84	0	30	0.23	1.00	0.40	0.37	0.77
	CodeBERT$_{wo-gt}$	36	73	2	28	0.33	0.95	0.46	0.49	0.83
Yes	**BERT$_{gt}$**	68	41	5	25	0.66	0.92	0.69	0.77	0.83
	ODS$_{gt}$	30	79	0	30	0.8	0.94	0.43	0.43	0.81
	CodeBERT$_{gt}$	74	35	3	27	0.68	0.96	**0.73**	**0.77**	**0.89**

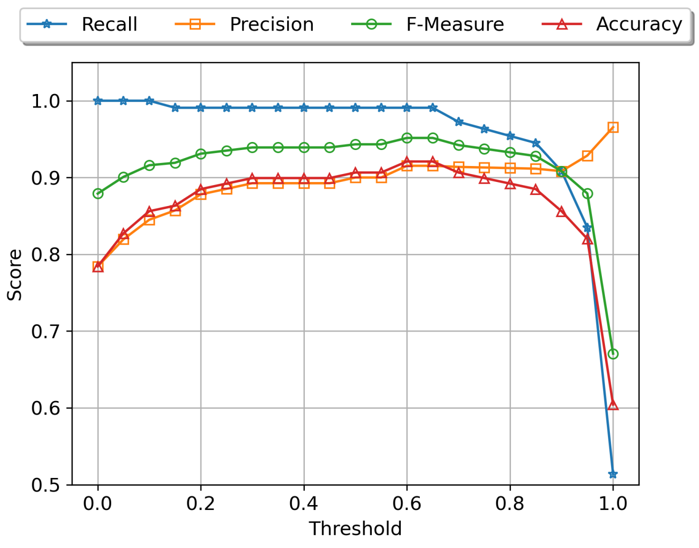


Fig. 7: The performance of Invalidator with different classification thresholds on evaluation set

5.3.3 Threshold Sensitivity

[RQ3.1: The impact of threshold sensitivity on the performance of Invalidator]
Recall that INVALIDATOR uses a threshold, which ranges from 0 to 1, to classify whether a patch is overfitting based on a prediction score produced by Machine Learning predictors as defined in Section 4.2.4. In this sub-question, we investigate the performance of INVALIDATOR in terms of *Recall*, *Precision*, *F-Measure* and *Accuracy* with different classification threshold in range (0, 1). The impact of the classification threshold on the performance of our approach is illustrated in Figure 7.

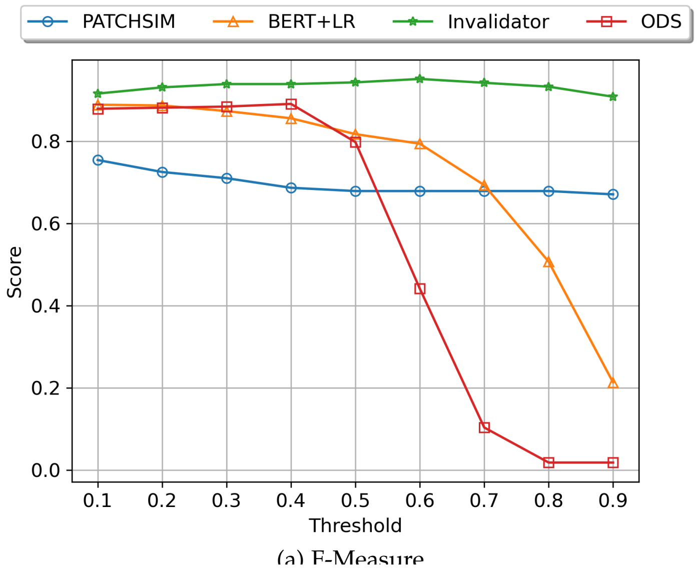
We can see that the *Recall* holds steady at around *1.0* when the classification threshold in range of (0.0, 0.65), then slightly decreases to about 0.94 (at threshold of 0.85) before dropping to about *0.53* at maximum threshold of 1.0. On the contrary, as the classification threshold increases, the *Precision* gradually increases from 0.79 to 0.97. Notably, INVALIDATOR's precision is always higher than 0.8 and around 0.9 most of thresholds. These results indicate that the assessment of INVALIDATOR is reliable.

With respect to *Accuracy* and *F-Measure*, the performance of INVALIDATOR shares a similar trend on these metrics according to the variation of the classification threshold. In details, *Accuracy* and *F-Measure* consistently increase from 0.79 and 0.89 to 0.92 and 0.95, respectively, when the threshold increases from 0.0 to about 0.6. Then, these metrics slightly decrease to 0.84 of *Accuracy* and 0.89 of *F-Measure* at the threshold of 0.9 before dropping to below 0.7 at maximum threshold of 1.0.

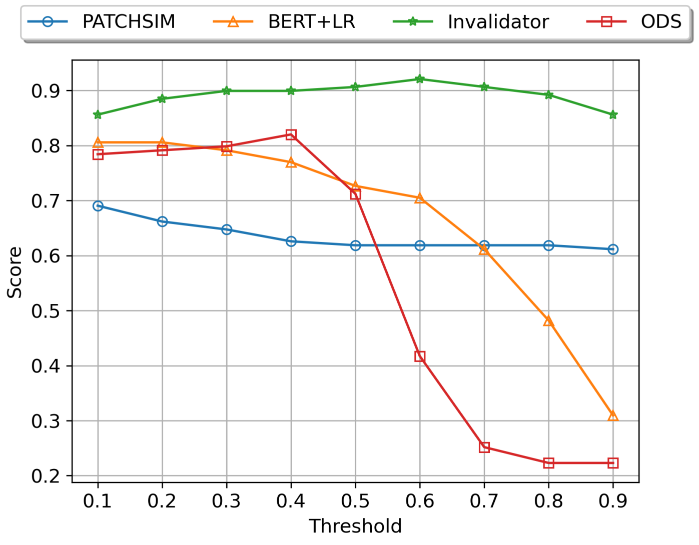
In conclusion, the results indicate that the classification threshold, although affects the *Recall* and *Precison*, has a limited impact on *F-Measure* and *Accuracy*. This is important for practitioners to choose a suitable threshold according to their demand without affecting the discriminant capability, which is reflected by *F-Measure* and *Accuracy*, of APAC techniques.

Answers to RQ3.1: Despite the change of *Precision* and *Recall*, INVALIDATOR still achieves promising overall performance, i.e., *F-Measure* and *Accuracy* at above 0.8, over a large range of classification threshold, i.e., (0.1 - 0.9), on both validation and evaluation set.

[RQ3.2: Invalidator vs. Existing threshold-dependence techniques]
In this sub-question, we compare the performance, reflected by *F-Measure* and *Accuracy)*, of four threshold-dependence techniques consisting of INVALIDATOR, ODS [23], BERT+LR [24] and PATCHSIM [21] with nine different threshold in range of (0.1, 0.9). The impact of the classification threshold on the performance of threshold-dependence techniques is illustrated in Figure 8. The results yield two main findings. First, the classification threshold has a limited impact on the performance of INVALIDATOR and PATCHSIM. Meanwhile, BERT+LR and ODS only achieve good performance in threshold range of (0.1, 0.4) before witnessing a significant decrease of both *F-Measure* and *Accuracy* when the threshold increases from 0.4 to 0.9. The finding indicates that INVALIDATOR and PATCHSIM are more stable than BERT+LR and ODS with respect to the variation of classification threshold. Second, INVALIDATOR, with an arbitrary threshold, performs better than the best result of each baseline. The finding indicates



(a) F-Measure



(b) Accuracy

Fig. 8: The performance of Invalidator, ODS, BERT+LR and PATCHSIM with different classification thresholds

that INVALIDATOR is the most effective technique among threshold-dependence APAC approaches.

Answers to RQ3.2: INVALIDATOR is the most effective and stable technique among threshold-dependence APAC approaches.

5.3.4 Ablation Study

[RQ4.1: The impact of semantic-based and syntactic-based classifier on the performance of Invalidator] In this experiment, we evaluate the relative contribution of INVALIDATOR's semantic versus structural classifier for patch correctness assessment. Table 6 shows the results of our experiments. INVALIDATOR$_{sem}$, INVALIDATOR$_{syn}$ refer to semantic and syntactic-based classifier, respectively. In the ablation study, we can observe that INVALIDATOR without these classifiers suffer from different degrees of performance loss. Specifically, removing INVALIDATOR$_{syn}$ leads to a decrease of 26% and 23% in terms of Accuracy and F1; meanwhile without INVALIDATOR$_{sem}$, INVALIDATOR's performance also drops by 11% and 8%, respectively. Also, we can see that our syntactic-based classifier shows a better performance than our semantic-based classifier. This is mainly because our semantic-based classifier can only detect 56 overfitting patches compared to 74 of our syntactic-based classifier. One potential reason behind the phenomena is that our semantic-based classifier depends on our current test suite, which may be an incomplete and invariant generator, i.e., Daikon. Therefore, though our semantic-based classifier can reveal hidden behavior differences between the APR-patched and ground truth programs to detect overfitting patches, its effectiveness can still be bounded by the abovementioned factors. However, the semantic-based classifier is still important for our approach to dealing with the threshold sensitivity of syntactic-based classifiers. Indeed, our semantic-based classifier is threshold-independent, allowing its performance to be considered a lower bound for the performance of INVALIDATOR. Therefore, INVALIDATOR still can work well with a strict classification threshold, making INVALIDATOR become the most stable technique among threshold-dependence APAC approaches, as we can see in the RQ 3.2. These results suggest that both semantic and syntactic-based classifiers are essential for the performance of INVALIDATOR.

Answers to RQ4.1: Our ablation study shows that both semantic and syntactic-based contribute to the effectiveness of INVALIDATOR.

[RQ4.2: The impact of invariant granularity on the performance of Invalidator]
In this sub-question, we investigate the performance of our semantic classifier with invariant inferred from two different granularities: buggy methods and executed methods, i.e., methods executed by test cases. As shown in Table 7, the invariants inferred from executed methods can boost the performance of our semantic classifier in APAC by 28% at *Accuracy* and 31% at *F-Measure*. The key reason for the improvement is that behavioral differences between APR-generated patches and correct patches exist in methods called by a statement of buggy methods. Hence, the supplement of invariant inferred from all executed methods helps our semantic classifier to detect more overfitting patches.

Answers to RQ4.2: The supplement of invariant inferred from all executed methods helps our semantic classifier boost the performance by 31% at *Accuracy* and 35% at *F-Measure*

[RQ4.3: The impact of different overfitting rules on the performance of Invalidator]
In this sub-question, we investigate the impact of each overfitting rule on the performance of our semantic classifier.
As shown in Figure 9 the *Overfitting-1* and *Overfitting-2* contributes 24 and 42 overfitting patches, respectively, among 56 patches detected by our semantic classifier. Moreover, there are 8 overfitting patches violating both overfitting rules. The results indicate that *Overfitting-2* rule contributes to our semantic classifier much more than *Overfitting-1*.

TABLE 6: Ablation Study. The **Invalidator**$_{Syn}$ and **Invalidator**$_{Sem}$ denotes INVALIDATOR's syntactic and semantic classifiers, respectively. The bold number denotes the better result in each evaluation metrics

Techniques	TP	FN	FP	TN	Recall	Precision	Accuracy	F1
Invalidator	86	23	3	27	**0,79**	**0,97**	**0,81**	**0.87**
-w/o **Invalidator**$_{Syn}$	56	53	2	28	0,51	**0,97**	0,60	0.67
-w/o **Invalidator**$_{Sem}$	74	35	3	27	0,68	0,96	0,73	0.80

TABLE 7: Overall performance of *Invalidator's semantic classifier* with different invariant granularity: buggy methods and executed methods. The bold number denotes the better result in each evaluation metrics

Granularity	Recall	Precision	F1	Accuracy
Buggy methods	0,35	0,95	0,51	0,47
Executed methods	**0,51**	**0,97**	**0,67**	**0,60**

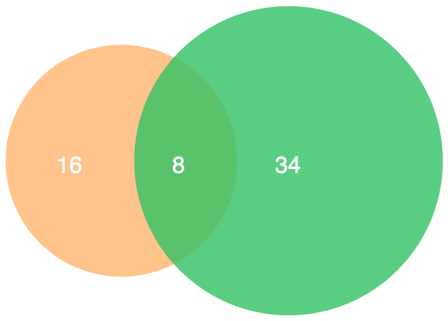


Fig. 9: The impact of overfitting rules on the performance of *Invalidator's semantic classifier*

Answers to RQ4.3: *Overfitting-2* rule contributes to INVALIDATOR much more than *Overfitting-1* (42 vs. 24 overfitting patches).

6 DISCUSSION

6.1 Time efficiency

With respect to time efficiency, we limit 5 hours for invariant inference for each patch in our dataset. In case invariants of a patch cannot be generated on time, we directly pass the patch to our syntactic classifiers. Meanwhile, for assessing the correctness of 139 patches in our evaluation dataset, i.e., Xiong et al. dataset INVALIDATOR took 15.5 hours (i.e., about 7 minutes for each patch). The results show that assessment time of INVALIDATOR is reasonable but the invariant inference is time-consuming. However, invariant inference is partially reusable as users can reuse the generated invariants for buggy and patched programs for each patch. Moreover, users can change the time limit for invariant inference if they only have the limited time budget. However, even in the worst case, the performance of INVALIDATOR will only drop to the performance of syntactic classifier, which still outperform the state-of-the-art baselines. Thus, we would like to leave the limitation of invariant inference for future works.

6.2 Threats to validity

External validity. Threats to external validity correspond to the generalizability of our findings. Our study considers 885 patches generated from 21 popular APR techniques. This may not represent all APR techniques thus may affect the generalizability of our study. We tried to mitigate this risk by selecting a data set that is commonly used for patch validation in the APR community [21], [17], [44], [60]. Another threat to external validity is that patches in our dataset are only generated for a dataset Defects4J. This may not represent all bugs in real-world projects and thus may affect the generalizability of our findings. Unfortunately, beside Defects4j, there is only one labeled dataset for patch correctness assessment, i.e., QuixBugs.QuixBugs, however, only contains small programs (approximately 35 lines of code on average) that implement basic algorithms such as Depth First Search or Knapsack. These programs differ from our focus in the paper: industrial programs. Meanwhile, obtaining ground-truth labels for patches for industrial programs datasets such as Bears and Bugs.jar requires extensive human efforts [17]. Therefore, we would like to leave the evaluation for future work.

Internal validity. Threats to internal validity refer to possible errors in our implementation and experiments. To mitigate this risk, we have carefully re-checked our implementation and experiments.

Construct validity. Threats to construct validity correspond to the suitability of our evaluation. The main threat is that the correctness of the patches may be subjectively biased because they are manually labeled by author annotation as mentioned in Section 2.1. To mitigate this risk, we collected the classification results, which come from reliable sources and are widely used by the community.

7 RELATED WORK

7.1 Automated Program Repair

Our study investigates patches generated by several popular APR techniques, including GENPROG [14], KALI [2], NOPOL [4], HDREPAIR [3] and ACS [7]. GENPROG and KALI are heuristic-based techniques which construct a search space by using mutation operations then leverage genetic programming to find the solution. NOPOL uses Satisfiability Modulo Theories to synthesize repair for buggy conditional statements. HDREPAIR mines historical bug fix patterns to guide the heuristic search. ACS attempts to generate high-quality repairs for buggy conditional statements by using historical fix templates. Beyond these techniques, recently, CAPGEN [61], SIMFIX [62], FIXMINER [28], and TBAR [10] have been proposed to fix bugs automatically based on frequent fix patterns. Other approaches (e.g., SEQUENCER [29], DLFIX [63], COCONUT [64]) propose to generate patches by using deep learning models.

7.2 Overfitting Problem

Early APR techniques widely leverage test suites, which are often practically weak and incomplete, as an oracle to guarantee patch correctness. This leads to the overfitting problem, in which APR-generated patches pass the validation test suite but are still incorrect [12]. Many APR techniques, e.g., GENPROG [14], RSREPAIR [27], AE [32], ANGELIX [6] have been shown to suffer from the overfitting issue [2], [13].

The overfitting problem has progressively been an important challenge in APR. Monperrus et al. criticized that the conclusiveness of techniques that keep patches and their correctness labels private is questionable [65]. Le et al. also suggested making publicly available to the community authors' evaluation on patch correctness [17]. Since then, APR techniques have publicly released their results and labels of APR-generated patches. Authors of APR techniques often assess patch correctness by either using: (1) an independent test suite different from the test suite used for repair to test the generalizability of the generated patches [17], or (2) manual inspection to compare APR-generated patches with the ground truth [61], [28], [10], [66]. Le et al. show that automated validation via independent test suite is less effective than manual validation, but there is a potential risk of human bias when using manual validation [17]. Also, manual validation requires repetitive and expensive tasks, which automated validation can complement.

In this work, we use a data set of 1028 APR-generated patches for large real-world programs whose correctness labels have been released by recent popular work [67], [21], [17], [60], [44]. The correctness labels of the patches have been carefully examined by the community, e.g., researchers and independent developers, and thus serve as reliable ground truth labels to assess the effectiveness of APAC techniques that we will discuss next.

7.3 Automated Patch Correctness Assessment

To avoid the potential bias of manual patch validation, several techniques have been proposed to predict patch correctness automatically. These techniques can be categorized into different directions: (1) semantic-based APAC and (2) syntactic-based APAC. In this section, we briefly review well-known techniques for each direction.

7.3.1 Semantic-based APAC

With respects to semantic-based APAC, the closely related works to our work are DIFFTGEN proposed by Xin and Reiss [18] and RGT proposed by Ye et al. [43] Similar to our work INVALIDATOR, DIFFTGEN and RGT identifies patch correctness by relying on perfect oracles such as correct programs provided by human developers. To do so, DIFFTGEN uses EVOSUITE, an automated test generation technique to generate an independent test suite from the developer-patched (ground truth) program. DIFFTGEN considers a APR-generated patch as overfitting if there are any behavioral differences between the APR-patched program and the ground truth program. The fundamental difference between these approaches and INVALIDATOR's semantic-based classfiier is that, instead of generating additional test cases, INVALIDATOR only uses the original test suite and infers program invariants to generalize the desired behaviors of the program under test. This way, INVALIDATOR generates more abstract program specifications in the form of program invariants to effectively guard against unintended behaviors of the programs under test.

Yu et al. [33] also generated additional test cases from the developer-patched program to detect two kinds of overfitting issues: incomplete fixing and regression introduction. However, their approach only works on semantic-based APR techniques while INVALIDATOR can identify overfitting patches generated by all APR approaches. Recently, Yang and Yang explored that majority of the studied plausible patches (92/96) expose different modifications of runtime behaviors captured by the program invariants, compared to correct patches [26]. However, this work does not propose any techniques to validate APR-generated patches. Based on findings of Yang and Yang, Ye et al. [68], and Wang et al. [44] have also used a simple heuristic based on DAIKON's invariants to identify patch correctness. These heuristics consider a patch as overfitting if it violates any invariants inferred from the developer-patched program. However, developers may add other functions which are unrelated to actual bugs, leading to redundant invariants. Hence, this overfitting behavior is weak and sensitive; that is the reason why they produce many false positives [68]. Meanwhile, Invalidator identifies patch correctness based on carefully designed overfitting behaviors by comparing invariants inferred from both buggy program and developer-patched program so that our technique essentially only produces a low false-positive rate, as shown in our evaluation.

Less relevant to our approach in this work are several techniques attempting to identify patch correctness without knowing perfect oracles. Yang et al. [69] proposed OPAD, which employs test-suite augmentation based on fuzz testing and uses the crash-free behavior as the oracle to detect overfitting patches. This approach, however, only identifies certain types of overfitting patches such as OPAD (as shown in Xiong et al.'s evaluation [43]). Xiong et al. [21] proposed PATCHSIM to heuristically identify patch correctness based on the similarity of test case executions. It first uses a test generation tool, i.e., RANDOOP, to generate new test inputs. It then automatically classifies the generated test cases into passing or failing based on the similarity on execution traces. Finally, it uses an enhanced test suite to determine whether a APR-generated patch is overfitting based on its behaviors on passing and failing test cases. Similar to DIFFTGEN, PATCHSIM requires the generation of external test cases while Invalidator only uses the original test suite and infers program invariants to generalize the desired behaviors of the program under test.

7.3.2 Syntactic-based APAC

With respect to syntactic-based APAC, the closely related works to our work are BERT+LR proposed by Tian et al. [24]. BERT+LR assumes that correct codes are substantially different from incorrect codes. Toward this, BERT+LR leverages code representation techniques for differentiating the correct and overfitting patches. Particularly, BERT+LR first embeds a patched code and a buggy code into numerical vectors by using BERT [59] and using Logistic Regression to estimate the similarity between a patched code and a buggy code. Finally, a patch is considered incorrect/overfitting if the similarity is lower than a certain threshold. However, it is challenging to find a suitable threshold as the difference between correct and incorrect codes may differ between programs. Different from BERT+LR, we proposed considering the similarity of a patched program to its ground truth and buggy program. Particularly, our syntactic-based classifier relies on the intuition that a correctly patched code is more similar to the developer-patched code (ground truth) than a buggy code. This way, the similarity between a patched and ground-truth code can serve as a "soft threshold" that can be changed over different programs. As a result, our approach is more flexible than BERT+LR, leading to better performance, as seen in Section 5.3. Besides, our syntactic-based classifier has new syntactic features, i.e., CodeBERT [46], which has been demonstrated to be more effective than BERT features.

Other works rely on static analysis (i.e., static code features) to validate the generated patch, including ANTI-PATTERNS and ODS. Tan et al. [25] propose anti-patterns (i.e., specific static structures) to filter out overfitting patches. Ye et al. [23] leverage 4199 code features extracted from buggy code and generated patches as input to machine learning algorithms (i.e., logistic regression, KNN, and random forest) to rank potentially overfitting patches. However, this work requires manual hand-crafted features that were carefully (manually) engineered, while our approach automatically extracts features via a pretrained language model.

Different from the aforementioned approaches from both semantic and syntactic-based APAC, our approach leverages both semantic information, i.e., program invariants, and syntactic information, i.e., CodeBERT features, to reason about patch correctness.

8 Conclusion and Future Work

We proposed INVALIDATOR, a novel automated patch correctness assessment technique using semantic and syntactic reasoning via program invariants and program syntax. INVALIDATOR first infers program specifications in the form of program invariants, guarding against *correct* and *error* specification of a program under test. Based on the inferred specifications, INVALIDATOR effectively identifies whether a APR-generated patch is overfitting. In case the above invariant-based specification inference fails to determine an overfitting patch, INVALIDATOR further uses a machine learning model to estimate the probability that the APR-generated patch is overfitting. To do this, INVALIDATOR first uses CODEBERT, a well-known pretrained model of code, to represent language semantic of program syntax via a vector of numbers, then measures syntactic differences between APR-generated patches and its buggy and correct version. Based on syntactic differences, INVALIDATOR uses a trained model from labelled patches to estimate the likelihood of a APR-generated patch being overfitting. We compared INVALIDATOR against state-of-the-art automated patch correctness assessment techniques from a popular data set of 885 APR-generated patches for large real-world projects in DEFECTS4J. Experiment results showed that INVALIDATOR outperforms state-of-the-art baselines.

In future work, we plan to extend INVALIDATOR with other groundtruth such as the original version of program before applying a bug-inducing commit. Moreover, the effectiveness of INVALIDATOR demonstrates that program invariants can effectively capture the runtime behaviors of the program. Therefore, another potential direction may be finding a way to take advantage of the program invariants in enhancing automated program repair directly. Finally, we plan to integrate INVALIDATOR as a part of training process to further improve learning-based program repair as inspired from Ye et al. [70].

9 Data Availability

Invalidator are publicly available at https://github.com/thanhlecongg/Invalidator. All of materials including implementation, datasets and experimental results are also published via https://doi.org/10.5281/zenodo.7475916

References

[1] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 2013, pp. 802–811.

[2] 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 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[3] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.

[4] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[5] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[6] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 691–701.

[7] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceeding of the 39th International Conference on Software Engineering*. IEEE, 2017, pp. 416–426.

[8] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax-and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 593–604.

[9] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, "Lsrepair: Live search of fix ingredients for automated program repair," in *Proceeding of the 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 658–662.

[10] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[11] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *Proceeding of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 269–278.

[12] 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 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532–543.

[13] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, vol. 23, no. 5, pp. 3007–3033, 2018.

[14] 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*. IEEE, 2009, pp. 364–374.

[15] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: a human study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 64–74.

[16] D. C. Kozen, "Rice's theorem," in *Automata and Computability*. Springer, 1977, pp. 245–248.

[17] D. X. B. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *Proceeding of the 41st International Conference on Software Engineering*. IEEE, 2019, pp. 524–535.

[18] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation." in *ISSTA*, vol. 17, 2017, pp. 226–236.

[19] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[20] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[21] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 789–799.

[22] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[23] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, 2021.

[24] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 981–992.

[25] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 727–738.

[26] B. Yang and J. Yang, "Exploring the differences between plausible and correct patches at fine-grained level," in *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. IEEE, 2020, pp. 1–8.

[27] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[28] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, pp. 1–45, 2020.

[29] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[30] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "Jfix: semantics-based repair of java programs via symbolic pathfinder," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 376–379.

[31] 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*. IEEE, 2009, pp. 364–374.

[32] 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)*. IEEE, 2013, pp. 356–366.

[33] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system," *Empirical Software Engineering*, vol. 24, no. 1, pp. 33–67, 2019.

[34] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.

[35] Z. Y. Ding, Y. Lyu, C. Timperley, and C. Le Goues, "Leveraging program invariants to promote population diversity in search-based automatic program repair," in *Proceeding of the International Workshop on Genetic Improvement (GI)*. IEEE, 2019, pp. 2–9.

[36] T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks, "Counterexample-guided approach to finding numerical invariants," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 605–615.

[37] D. Ishimwe, K. Nguyen, and T. Nguyen, "Dynaplex: analyzing program complexity using dynamically inferred recurrence relations." *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–23, 2021.

[38] T. C. Le, T. Antonopoulos, P. Fathololumi, E. Koskinen, and T. Nguyen, "Dynamite: dynamic termination and non-termination proofs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[39] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. Proceedings of the 26th International Symposium on Software Testing and Analysis. New York, NY, USA: Association for Computing Machinery, 2016, p. 177–188. [Online]. Available: https://doi.org/10.1145/2931037.2931049

[40] D. Gopinath, H. Converse, C. Pasareanu, and A. Taly, "Property inference for deep neural networks," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 797–809.

[41] T.-D. Nguyen, T. Le-Cong, T. H. Nguyen, X.-B. D. Le, and Q.-T. Huynh, "Toward the analysis of graph neural networks," in *2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2022, pp. 116–120.

[42] P. Sagdeo, N. Ewalt, D. Pal, and S. Vasudevan, "Using automatically generated invariants for regression testing and bug localization," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 634–639.

[43] H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *Empirical Software Engineering*, vol. 26, no. 2, pp. 1–38, 2021.

[44] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 968–980.

[45] 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*. Springer, 2008, pp. 337–340.

[46] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[47] S. WANG and J. JIANG, "A compare-aggregate model for matching text sequences.(2017)," in *ICLR 2017: International Conference on Learning Representations, Toulon, France, April 24-26: Proceedings*, 2017, pp. 1–15.

[48] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *Proceeding of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 425–436.

[49] T. G. Nguyen, T. Le-Cong, H. J. Kang, X.-B. D. Le, and D. Lo, "Vulcurator: a vulnerability-fixing commit detector," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1726–1730.

[50] T. Le-Cong, H. J. Kang, T. G. Nguyen, S. A. Haryono, D. Lo, X.-B. D. Le, and Q. T. Huynh, "Autopruner: transformer-based call graph pruning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 520–532.

[51] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems,*, 2017, pp. 5998–6008.

[52] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020, pp. 518–529.

[53] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 23th International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[54] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *Proceeding of the 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 215–224.

[55] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *2012 34th International Conference on Software Engineering*. IEEE, 2012, pp. 386–396.

[56] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *Proceeding of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 705–716.

[57] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: a discriminative pattern mining approach," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009, pp. 557–566.

[58] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *Proceeding of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[59] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *NAACL-HLT*, pp. 4171–4186, 2018.

[60] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon, "On the efficiency of test suite based program repair," in *International Conference on Software Engineering*, 2020.

[61] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering*. IEEE, 2018, pp. 1–11.

[62] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 298–309.

[63] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

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

[65] M. Monperrus, "A critical review of" automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 234–242.

[66] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 1–12.

[67] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[68] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," in *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. IEEE, 2019, pp. 1–10.

[69] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 831–841.

[70] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1506–1518.