Invalidator: Automated Patch Correctness Assessment Via Semantic and Syntactic Reasoning

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

***Abstract*—Automated program repair (APR) faces the challenge of test overfitting, where generated patches pass validation tests but fail to generalize. Existing methods for patch assessment involve generating new tests or manual inspection, which can be time-consuming or biased. In this paper, we propose a novel technique, Invalidator, to automatically assess the correctness of APR-generated patches via semantic and syntactic reasoning. Invalidator leverages program invariants to reason about program semantics while also capturing program syntax through language semantics learned from a large code corpus using a pre-trained language model. Given a buggy program and the developer-patched program, Invalidator infers likely invariants on both programs. Then, Invalidator determines that an APR-generated patch overfits if: (1) it violates correct specifications or (2) maintains erroneous behaviors from 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 threefold. First, Invalidator leverages both semantic and syntactic reasoning to enhance its discriminative 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 program behaviors. Third, Invalidator is fully automated. Experimental results demonstrate that Invalidator outperforms existing methods in terms of Accuracy and F-measure, correctly identifying 79% of overfitting patches and detecting 23% more overfitting patches than the best baseline.**

***Index Terms*—Automated patch correctness assessment, automated program repair, code representations, overfitting problem, program invariants.**

I. Introduction

Automated program repair (APR) is a promising approach to alleviate the onerous burden on developers to manually fix bugs. Over the years, various APR techniques have been proposed [3], [15], [21], [23], [26], [27], [31], [32], [35], [37], [40], [48], [64], [67], [68], [73], [75], with several breakthroughs that inspired potential practical adoption of APR. Notably, Facebook has recently deployed SapFix [38], the first-ever industrial-scale automatic bug-fixing system, for suggesting fixes to developers in real-world products. Despite these recent successes, APR still suffers from a major challenge, namely *test overfitting* [24], [45], [51], in which a generated patch may pass all test cases but still fails to generalize to the intended behaviors of the program. According to Qi et al. [48] 98% of the plausible patches generated by GenProg [27] are overfitting.

Detecting overfitting patches is one key challenge that is important not only to ensure fair comparisons between APR techniques but also to enable the practical adoption of APR by developers. Often, one APR technique claims to be better than others only solely in terms of the number of bugs for which it can generate “correct” patches. Furthermore, recent research suggested that low-quality patches may negatively affect developers’ performance [54]. A fundamental question then arises,

> *“How can we determine whether a patch is correct?”*

Unfortunately, even with the availability of the ground truth (developer-patched) program, it is difficult to determine whether an APR-patched program is correct. That is because, 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 [17].

Recent approaches in automated program repair (APR) have explored various techniques to assess the correctness of APR-generated patches in comparison to developer-written patches as ground truth. These approaches include the use of test-suite augmentation and human manual inspections. While being human manual inspections are effective [18], they are prone to human biases, are expensive, and require manual, repetitive tasks. On the other hand, test-suite augmentation approaches, such as those proposed by Xin et al. [65] and Xiong et al. [67], are fully automated, but they require the generation of at least one test case to observe behavioral differences between the APR-patched program and the ground truth program. However, a recent study [18] has shown that test-suite augmentation approaches are often ineffective as the search space for bug-witnessing test cases can be large. Even state-of-the-art test case generation techniques, such as Randoop [46] and DiffTGen [65], can only identify 22% of the overfitting patches generated by APR

Manuscript received 9 May 2022; revised 6 March 2023; accepted 6 March 2023. Date of publication 10 March 2023; date of current version 14 June 2023. This work was supported by the Australian Research Council’s Discovery Early Career Researcher Award under Grant DE220101057. Recommended for acceptance by D. Bianculli. *(Corresponding author: Quyet-Thang Huynh.)*

Thanh Le-Cong and Xuan Bach D. Le are with the School of Computing and Information Systems, The University of Melbourne, Parkville, VIC 3010, Australia (e-mail: congthanh.le@student.unimelb.edu.au; bach.le@unimelb.edu.au).

Duc-Minh Luong, Nhat-Hoa Tran, Bui Quang-Huy, and Quyet-Thang Huynh are with the School of Information and Communication Technology, Hanoi University of Science and Technology, Ha Noi 100000, Vietnam (e-mail: minh.ld176821@sis.hust.edu.vn; hoant@soict.hust.edu.vn; 20202517M@student.hust.edu.vn; thanghq@soict.hust.edu.vn).

David Lo is with the School of Computing and Information Systems, Singapore Management University,, Singapore 188065 (e-mail: davidlo@smu.edu.sg).

Digital Object Identifier 10.1109/TSE.2023.3255177

tools [18]. Furthermore, Xin et al. [65] reported that in many cases, the state-of-the-art test generator EvoSuite [9] failed to generate any test methods that exercise the code changes introduced in the generated patches, and thus failed to identify behavioral differences between the patches and the ground truth.

In this paper, we introduce a novel technique called Invalidator that combines semantic and syntactic reasoning to automatically assess the correctness of patches generated by APR techniques. Invalidator leverages program invariants to reason about program semantics and pre-trained language models to capture program syntax by learning language semantics from a large code corpus. Similar to other automated patch correctness assessment (APAC) techniques, Invalidator utilizes behavioral discrepancies between the APR-patched and ground truth programs to determine the patch's correctness. However, conceptually, Invalidator is different from the strategy employed by existing APAC techniques such as DiffTGen [65], PatchSim [66], or Randoop [18], [46]. These techniques generate new tests to augment the current test suite, in which each test generates one execution. As a result, 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 incorporating syntactic reasoning to enhance its effectiveness. We describe the details of the semantic and syntactic reasoning in Invalidator below.

Given an 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 the *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 the *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 [7], 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 the 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, namely CodeBERT to extract source syntactic features from the 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 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 [74], ODS [71], BERT+LR [55], PatchSim [66], DiffTGen [65], Anti-patterns [53], Daikon [69]. Experiment results showed that Invalidator correctly classified 79% of 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 versus 0.68) and 19% (0.87 versus 0.76) in terms of *Accuracy* and *F1-score*, 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 assess 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 that rely on program invariants to assess APR-generated patches. Our empirical evaluation demonstrated that these rules can effectively identify 51% of overfitting patches with a precision of 97%.
- We proposed using syntactic reasoning from the program source code to augment semantic reasoning from the two aforementioned overfitting rules. Our empirical evaluation showed that syntactic reasoning can boost the performance of our approach by 35% and 30% in terms of Accuracy and F1-score, respectively.
- We conducted experiments on 885 machine-generated patches for the Defects4J benchmark. The experiment results showed that the unique combination of syntactic and semantic reasoning empowers Invalidator to achieve substantial improvements (i.e., 19% and 14% for Accuracy and F1-score, respectively) over state-of-the-art baselines.

The rest of this paper is organized as follows: Section II provides the background information on the overfitting problem, APAC, and program invariants. Section III presents a motivating example for our approach, followed by Section IV which describes our approach in detail. Section V presents our experimental setup and results. In addition, Section VI discusses the efficiency, potential applications, and threats to validity of our approach. Section VII provides an overview of related work in this area. Finally, in Section VIII, we conclude our paper and discuss future work.

II. Background

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

A. Automated Program Repair

Program Repair. Given a buggy program and a set of test cases in which there exists 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 [27], random search [47], or learning algorithms such as data mining and machine learning, e.g., [23], [16], [3], and [67], to apply mutations and evolve the buggy program until they find a patch passing the test suite. Semantics-based repair techniques, e.g. ANGELIX [40], S3 [21], JFIX [20], 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 VII).

Overfitting. One primary challenge in automated program repair (APR) is that APR-generated patches can be tests-adequate but may not generalize. This phenomenon, known as the 'test overfitting' problem, refers to situations where APR-generated patches successfully pass all test cases but are not semantically correct, as demonstrated in prior work [24], [45], [48], [51]. Early APR techniques utilized an existing test suite as an oracle to evaluate the correctness of generated patches [27], [62]. Specifically, a patch is considered correct if it successfully passes all test cases and incorrect otherwise [27], [62]. However, recent studies [48], [51] showed that this assessment method is insufficient to ensure the correctness of generated patches, as the test suite used for evaluation is often incomplete. Through manual analysis, Qi et. al. [48] have shown that the majority of patches generated by search-based APR techniques, such as GENPROG [27], AE [61], and RSREPAIR [47] exhibit overfitting. Similarly, through automated evaluation, Le et al. [24] also have reported that semantic-based repair techniques, such as ANGELIX [40] are no exception to the overfitting issue.

Automated Patch Correctness Assessment. Recently, researchers have often adopted one of two approaches for assessing the correctness of program repairs: (1) manual annotation, where the authors of repair techniques manually judge the correctness of APR-generated patches by their own and competing approaches, or (2) automated assessment, where an independent test suite is used to automatically evaluate patch correctness. However, Le et al. [18] showed that while a manual annotation is more effective, it is also more expensive. In contrast, an automated assessment does not require a manual effort but is less effective [18]. Recent research efforts have been devoted to automated patch correctness assessment (APCA) [18], [65], [66]. Existing APCA techniques usually assume that ground truth patches are available for comparison [18], [65], [76]. For example, Xin et al. [65], and Le et al. [18] generate new test cases based on the program (i.e., ground truth) to identify overfitting patches. Our proposed technique also falls into this category, where we assume the availability of ground truth patches. However, unlike existing test-based approaches, our technique relies on program invariants to judge the correctness of APR-generated patches.

B. 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* [7]. 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 *hold for some executions, but perhaps not all* [1], [6]. Likely invariants can be automatically inferred from execution traces by dynamic invariant inference techniques which generalize from execution traces using invariant templates. Previous studies have demonstrated the effectiveness of likely invariants in various tasks including complexity analysis [12], [43], termination analysis [19], bug localization [2] and neural network analysis [10], [42].

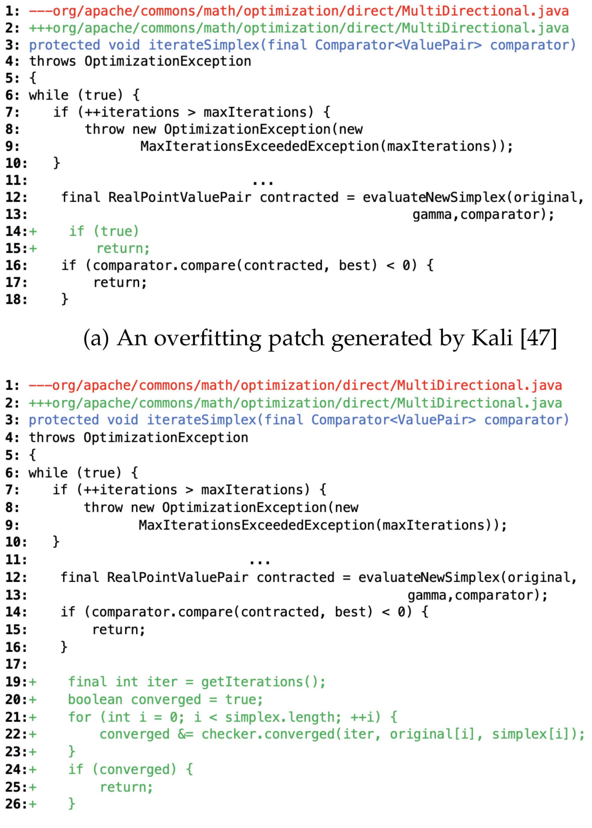
In this paper, we use Daikon [7] - a popular tool for mining likely invariants, as our dynamic invariant inference technique. Daikon observes the execution traces of programs and matches them against a set of templates to infer likely invariants that hold on all or most of the executions. From a large 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 [49].

III. Motivation

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

In this example, the `iterateSimplex()` method contains a bug that causes an endless loop (line 6 in Fig. 1(b)) when computing the next simplex of the multi-directional optimizer. To fix this issue, a developer-written patch was added (line 5 and lines 19-26 in Fig. 1(b)) that stops the loop once the algorithm converges. In contrast, the plausible patch generated by *Kali* [47] (Fig. 1(a)) inserts an early `return` at lines 14-15, causing the failing test case to plausibly pass. While the APR-patched program avoids the endless loop, it ignores the main algorithm, which requires many iterations to converge. Unfortunately, current state-of-the-art test-based automatic program repair techniques, such as RGT [74], DIFFTGEN [65], and RANDOOP [46], have difficulty identifying behavioral differences between the

[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 [47]

```
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.

APR-patched program and the ground truth [24] 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 for the algorithm to terminate after several iterations once it converges, regardless of the input. For the `iterateSimplex` method, an invariant `iterations > orig(iterations)` is inferred from both the buggy and correct versions of the program. This invariant indicates that the value of the `iterations` variable before calling the `iterateSimplex` method (denoted as `orig(...)`) is smaller than the value after execution. This variable measures the number of iterations executed by the algorithm and reflects the fact that the while loop (Line 4 in Fig. 1(a)) should execute until the multi-directional optimizer converges, which may require many iterations.

However, in the APR-patched program, the while-loop always terminates after the first iteration due to the code snippet `if (true) return;` (line 14-15 in Fig. 1(a)). Consequently, an invariant `iterations - orig(iterations) - 1 == 0` is obtained for the APR-patched program. This invariant indicates that the value of the `iterations` variable is always incremented by one after executing the `iterateSimplex` method. This behavior shows that the APR-patched program is overfitting and violates the intended behavior of the program, which requires varying numbers of iterations to converge under different inputs.

Our tool, INVALIDATOR, detects this overfitting behavior of the APR-patched program by comparing the invariant generated from the APR-patched program with that of the buggy and ground truth programs. Specifically, INVALIDATOR identifies that the APR-patched program maintains an invariant that never holds in the buggy and ground truth programs, indicating a behavioral divergence that could lead to errors.

IV. METHODOLOGY

Fig. 2 illustrates the workflow of INVALIDATOR. First, an APR-patched program is validated using a semantic-based classifier. During this phase, INVALIDATOR assesses the correctness of the patch based on *correct* and *error specifications*, which are inferred by analyzing the differences in behavior between the buggy program and its correct version. These specifications are captured using automatically-inferred program invariants. If the invariants inferred from the APR-patched program violate the correct specification or maintain the error specifications, the APR-patched program is considered *overfitting*. If the inferred specifications fail to identify an overfitting patch, INVALIDATOR uses a learning-based model that leverages syntactic reasoning to estimate the probability that the APR-patched program is overfitting. We provide further details about INVALIDATOR below.

A. Semantic-Based Patch Classifier

Fig. 3 illustrates how INVALIDATOR employs a semantic-based patch classifier to identify *overfitting patches*. First, INVALIDATOR constructs approximate specifications of the program under test by using a dynamic invariant inference tool, called DAIKON [7] (described in Section IV-A1). Then, based on the inferred specifications, INVALIDATOR automatically classifies whether an APR-patched program is overfitting (explained in Section IV-A2). We provide detailed explanations of each phase of INVALIDATOR below.

1) Invariant-Based Specification Inference: The goal of invariant inference is to derive specifications that can determine the correct and error behaviors of a program. INVALIDATOR employs invariants to approximate these specifications based on two key observations that enable the detection of behavioral differences, as outlined below:

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

Observation 2. Program invariants that exist only in the buggy program, but do not hold in the correct version, may represent the *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 construct them via invariant inference, and how they can be used to determine overfitting patches effectively.

The correct behaviors of a program are defined based on invariants in Definition 1. Correct behaviors 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 appears in both the buggy version and the correct version of

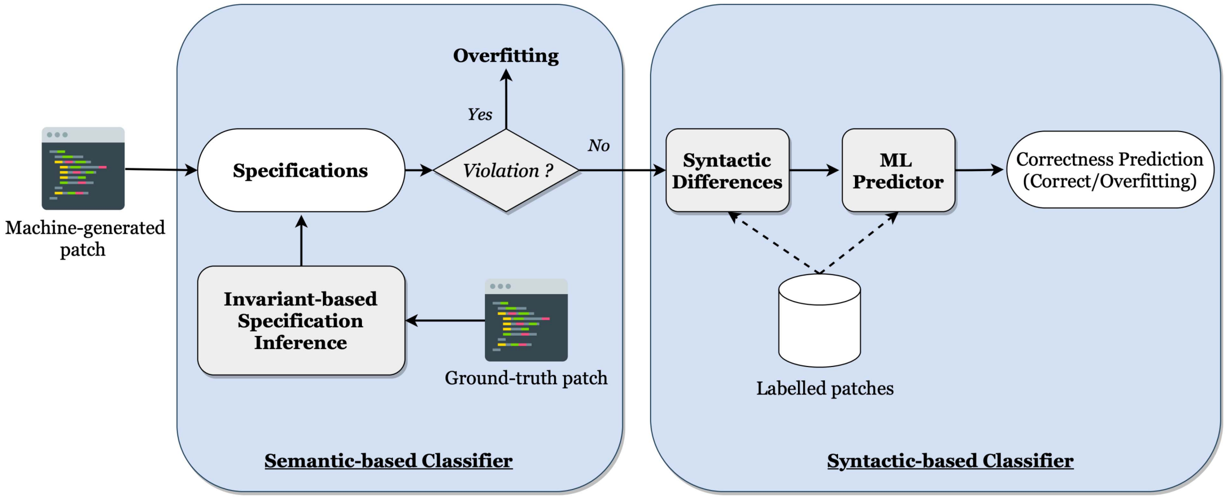

Machine-generated patch
Specifications
Violation ?
Yes
Overfitting
No
Invariant-based Specification Inference
Ground-truth patch
Semantic-based Classifier
Syntactic Differences
ML Predictor
Correctness Prediction (Correct/Overfitting)
Labelled patches
Syntactic-based Classifier

Fig. 2. The workflow of INVALIDATOR.

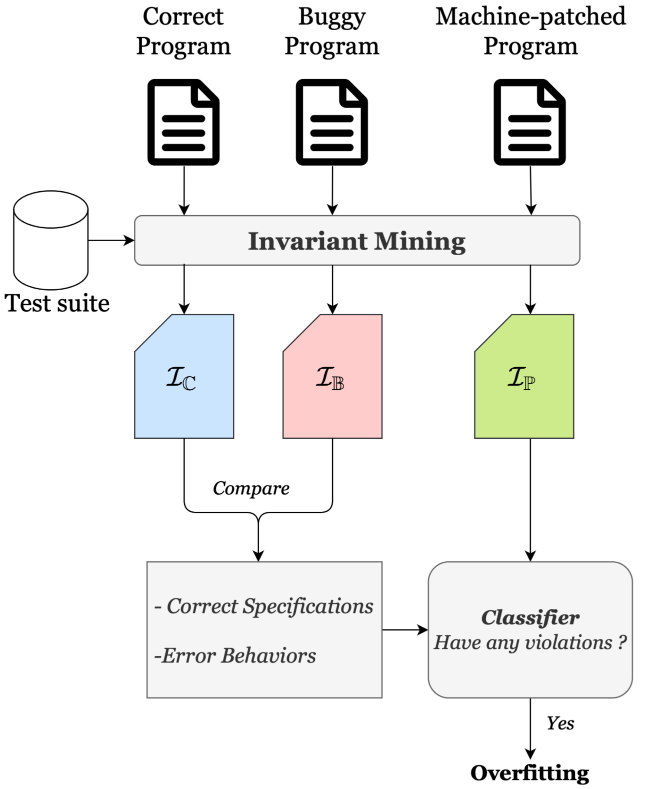


Fig. 3. APAC via invariant-based specification inference. $\mathcal{I}_{\mathbb{C}}$, $\mathcal{I}_{\mathbb{B}}$ and $\mathcal{I}_{\mathbb{P}}$ are sets of invariants inferred from correct program, buggy program and APR-patched program, respectively.

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 V).

Error behaviors are defined in Definition 2, and 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 $\mathcal{C} \models \mathbb{B}$ and $\mathcal{C} \models \mathbb{G}$, where X $\models$ Y denotes a semantic logical consequence relation in which all properties satisfying X also satisfy Y.

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 $\mathcal{E} \models \mathbb{B}$ and $\mathcal{E} \not\models \mathbb{G}$, where X $\models$ Y denotes a semantic logical consequence relation in which all properties satisfying X also satisfy Y.

Let us now explain how we construct the specifications that approximate the correct and error behaviors of a program as depicted in Fig. 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 the correct specification of a program while failing test cases $\mathbb{F}$ reflect the 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.

To construct these specifications, INVALIDATOR leverages DAIKON [7] to infer likely invariants based on the execution traces. DAIKON first captures runtime values of variables at specific points in a program, such as the points at which a method is entered or exited, and then it uses a set of templates that satisfy the runtime values to infer likely invariants, which are properties that hold over all of the executions. We refer to the invariants inferred from the passing and failing test cases of a program $prog$ as $\mathcal{I}^F_{prog}$ and $\mathcal{I}^P_{prog}$, respectively.

We use $\mathbb{B}$ and $\mathbb{G}$ to respectively denote the original buggy and correct (ground truth) versions of a program. To approximate the correct specification of $\mathbb{G}$, INVALIDATOR infers invariants from the passing test cases on $\mathbb{B}$ and $\mathbb{G}$, denoted as $\mathcal{I}^P_{\mathbb{B}}$ and $\mathcal{I}^P_{\mathbb{G}}$

Algorithm 1: Invariant-Based Specification Inference. $\mathbb{B}$ is the Original Buggy Program, $\mathbb{P}$ is an APR-Patched Program, and $\mathbb{G}$ is the Correct/Ground Truth Program by Developers.

```
Input:
  • $\mathcal{I}_{\mathbb{P}}^{P}$: invariant inferred from passing tests on $\mathbb{P}$
  • $\mathcal{I}_{\mathbb{P}}^{F}$: invariant inferred from failing tests on $\mathbb{P}$
  • $\mathcal{I}_{\mathbb{B}}^{P}$: invariant inferred from passing tests on $\mathbb{B}$
  • $\mathcal{I}_{\mathbb{B}}^{F}$: invariant inferred from failing tests on $\mathbb{B}$
  • $\mathcal{I}_{\mathbb{G}}^{P}$: invariant inferred from passing tests on $\mathbb{G}$
  • $\mathcal{I}_{\mathbb{G}}^{F}$: invariant inferred from failing tests on $\mathbb{G}$
Output: True: $\mathbb{P}$ is overfitting, False: Otherwise
$\mathcal{C} \leftarrow \mathcal{I}_{\mathbb{G}}^{P} \cap \mathcal{I}_{\mathbb{B}}^{P}$        ▷ Correct specification
$\mathcal{E} \leftarrow \mathcal{I}_{\mathbb{B}}^{F} \setminus \mathcal{I}_{\mathbb{G}}^{F}$        ▷ Error specification
foreach $inv$ in $\mathcal{C}$ do
    if $inv \notin \mathcal{I}_{\mathbb{P}}^{P}$ then
        return True
    end
end
foreach $inv$ in $\mathcal{E}$ do
    if $inv \in \mathcal{I}_{\mathbb{P}}^{F}$ then
        return True
    end
end
return False
```

respectively. The correct specification $\mathcal{C}$ of $\mathbb{G}$ is then constructed by intersecting the two sets of invariants $\mathcal{I}_{\mathbb{B}}^{P}$ and $\mathcal{I}_{\mathbb{G}}^{P}$ and taking the resulting invariants as an approximation for the correct specification of $\mathbb{G}$. To approximate the error specification $\mathcal{E}$ in $\mathbb{B}$, INVALIDATOR first infers invariants from the failing test cases on both $\mathbb{B}$ and $\mathbb{G}$, denoted as $\mathcal{I}_{\mathbb{B}}^{F}$ and $\mathcal{I}_{\mathbb{G}}^{F}$ respectively. The error specification $\mathcal{E}$ is then constructed by taking the invariants that are in $\mathcal{I}_{\mathbb{B}}^{F}$ but are not in $\mathcal{I}_{\mathbb{G}}^{F}$. In summary, the correct specification $\mathcal{C}$ represents the expected specification in $\mathbb{B}$ and $\mathbb{G}$, while the error specification $\mathcal{E}$ represents the behavioral difference of $\mathbb{B}$ compared to $\mathbb{G}$.

INVALIDATOR uses the constructed specifications $\mathcal{C}$ and $\mathcal{E}$ as inputs to its patch classifier, which we describe in Section IV-A2, to identify overfitting patches. Note that the INVALIDATOR classifier considers invariants inferred from all methods executed by a given test suite, rather than only invariants inferred from buggy methods (i.e., methods modified by human developers in the correct program) as in prior works [59], [69]. We discuss the effectiveness of the classifier using these two granularities in detail in Section V.

2) Patch Classifier: The patch classifier takes as input an 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 is based on two key observations:

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

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

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

- *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 an SMT solver, i.e., Z3 [4], 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 to be semantically equivalent by Z3; Z3 determines that the formulae $(a >= b \Rightarrow b <= a) \wedge (b <= a \Rightarrow a >= b)$ are satisfiable, and hence $a >= b$ is equivalent to $b <= a$.

3) Optimization Via Test Selection: Our observation is that not all test cases are relevant to the bug at hand. Therefore, before running DAIKON, INVALIDATOR performs *test selection*, as 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 optimize for running time without compromising its accuracy.

B. Syntactic-Based Patch Classifier

In case INVALIDATOR fails to reason about patch correctness via invariant-based specification inference (described in Section IV-A), INVALIDATOR resorts to estimating the probability that an APR-patched program is overfitting by measuring the syntactic proximity of the patch to the buggy and ground truth programs. Given an 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 pre-trained language model, i.e., CODEBERT [8], to extract syntactic features of these programs and then uses comparison functions [60] as distance measures to identify syntactic differences between them. Finally, INVALIDATOR uses a machine

Algorithm 2: Test Selection.

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

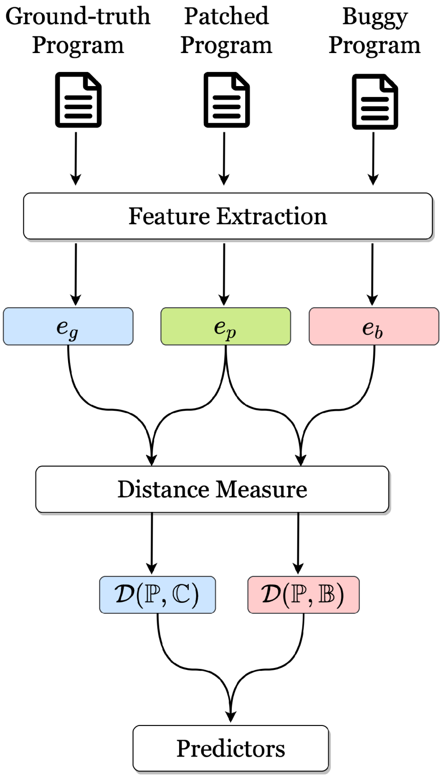


Fig. 4. Model architecture of the syntactic classifier. e_b, e_p, and e_c are representations of the buggy program, patched program, and ground truth program, respectively. $\mathcal{D}(\mathbb{P}, \mathbb{B})$, $\mathcal{D}(\mathbb{P}, \mathcal{G})$ are distances from patched program to buggy program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

learning model to predict patch correctness. Fig. 4 illustrates the classification pipeline of our syntactic-based classifier. Below, we explain each component of the pipeline in detail.

1) Feature Extraction: The feature extraction layers aim to extract embedding vectors (a.k.a. features) that represent the syntactic information of buggy, patched, and correct programs. To achieve this, we utilize CodeBERT [8], a powerful pre-trained model for general-purpose representations of source code that has demonstrated its effectiveness on various software engineering tasks [8], [25], [44], [78]. CodeBERT takes a code fragment as input and uses a tokenizer (i.e., the Roberta tokenizer) to tokenize the code into a sequence of tokens. It then passes the sequence through a pre-trained multi-layer bidirectional Transformer [58] to obtain a corresponding numerical vector. Specifically, given a code fragment, Invalidator employs CodeBERT to represent the code fragment as the vector defined as follows:

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

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

2) Distance Measure: The goal of the distance measure layers is to build the vectors that capture the syntactic differences between the APR-patched program, buggy program, and correct program. Inspired by prior works [11], [55], we leverage comparison functions [60] to represent various types of syntactic differences. The distance measure layers take as input the embedding vectors of the buggy program, patched program, and correct program (denoted by e_b, e_p, and e_c, respectively) and output the vectors that represent the syntactic difference of the APR-patched program compared to the 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 four comparison functions, consisting of cosine similarity, Euclidean distance, element-wise subtraction, and multiplication. 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 the patched program to the 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 the 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.

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

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

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

V. 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 V-A. Section V-B lists our research questions, followed by our findings in Section V-C.

A. Experimental Settings

1) Dataset: To evaluate the effectiveness of automated patch correctness assessment (APCA) techniques, we have collected a dataset of APR-generated patches whose correctness labels were manually identified by independent developers and researchers. We used 220 patches released by Xiong et al. [66] and 902 patches released by Wang et al. [59]. Following previous works [55], [65], we only consider patches from four widely-used projects in Defects4J: Chart, Time, Lang, and Math. This resulted in a dataset of 139 patches from Xiong et al.'s dataset and 666 patches from Wang et al.'s dataset. Additionally, to address data imbalance issues where very few APR-generated patches are labeled as correct, we supplemented the dataset with developer-written patches from the Defects4J dataset following [55]. This resulted in a dataset of 1028 patches, including 469 correct patches and 559 overfitting patches.

TABLE I
Dataset for Evaluating Automated Patch Correctness Assessment Techniques

Dataset	Correct patches	Overfitting patches	Total
Xiong et al. [66]	30	109	139
Wang et al. [59]	216	450	666
Defects4J [14]	223	0	223
Final dataset	**377**	**508**	**885**

TABLE II
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

We consider 666 patches from Wang et al.'s dataset and 223 developer's patches from Defects4J [14] as the training and validation set and 139 patches from Xiong et al. [66] as evaluation set following previous work [55], [55], [66], [71]. Note that, there may be duplication between Wang et al. 's dataset, Defects4J's patches, and Xiong et al.'s dataset. To avoid data leakage, we removed the duplicated patches from the training and validation set. Particularly, we removed a patch if it is syntactically equivalent to a patch in the evaluation set. As a result, we obtained 746 (out of 889) patches for the training and validation phase. This included 347 correct patches and 399 overfitting patches. We use 90% of these patches (90% × 746 = 671 patches) for training our learning model and the remaining 75 patches for validation.

Table I shows the details of the patches considered in our experiments, and Table II provides information on the characteristics of our training, validation, and evaluation datasets.

2) Evaluation Metrics: By using the dataset described in Section V-A1, 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 that of the ground truth labels. Specifically, we use standard metrics of classification problems [56], [57], *Recall* (2), *Precision* (3), *Accuracy* (4), and *F1-score* (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.

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

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

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

$$\textbf{F1} = \frac{2 \text{ x } Recall \text{ x } Precision}{(Precision + Recall)} \tag{5}$$

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 [54]. 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 [71].

However, the comparison of APAC techniques that relies only on *Recall* or *Precision* may be incomplete. For example, an 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 *F1-score* and *Accuracy* as additional evaluation metrics to measure the performance of APAC techniques following previous studies [29], [74], [79]. *F1-score* 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.

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

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 [34], [77]. In our paper, *AUC* is essential to compare the performance of syntactic-based classifiers.

3) Implementation Details: For INVALIDATOR, we implement the proposed approach using Python programming language. For the 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 the 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 [55], [66]. Note that, we tune our classification threshold on an independent validation set (see details in Section V-A1) instead of the evaluation set as prior works [55], [66] to avoid overfitting. We also investigate the impact of the threshold on the performance of INVALIDATOR in Section V.

With respect to baseline techniques, we collect results of ODS, PATCHSIM, ANTI-PATTERNS, and BERT + LR from prior works [55], [66], [71]. 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.

B. 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 V-A1) in terms of *Precision*, *Recall*, *Accuracy*, and *F1-score*. Then, we compare our approach to state-of-the-art baseline techniques, namely:

- RGT: Ye et al. [74] proposed to use a testing procedure, named Random Testing with Ground truth (RGT) [50], 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 an APR-patched program, RGT considers the program as *overfitting*.
- ODS: Ye et al. [71] 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. [55] proposed a learning-based APAC technique that utilizes BERT [5] and Logistic 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. [66] 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. [65] proposed an 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 [53], 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 [69] discovered that the majority of overfitting patches exhibit distinct

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

runtime behaviors captured by the invariants generated by GT-INVARIANT [7]. Building on this insight, Wang et al. [59] propose a straightforward heuristic that considers an APR-generated patch as *overfitting* if any of its inferred invariants differ from those of the correct program. In this paper, we adopt their technique and refer to it 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 *F1-score* 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 employs a classification threshold to determine 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 yielded the highest precision on the validation dataset for INVALIDATOR. In this research question, we investigate the impact of threshold sensitivity on INVALIDATOR's performance. To this end, we conduct experiments to address two sub-questions:

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

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

This research question aims to analyze the contribution of different components of INVALIDATOR to its overall performance. Firstly, we investigate the impact of semantic and syntactic classifiers on INVALIDATOR's performance. Next, we examine the impact of design choices for each component, including the granularity of invariants and overfitting rules, on the performance of INVALIDATOR. Specifically, we address three sub-questions as follows:

- **RQ4.1:** *How do semantic and syntactic classifiers affect the performance of our approach?* INVALIDATOR contains two main components: semantic and syntactic classifiers. In this research question, we perform an ablation study by dropping each classifier to evaluate the contribution of each classifier to INVALIDATOR's performance.
- **RQ4.2:** *Does using invariants inferred from executed methods improve the performance of* INVALIDATOR*compared to using invariants inferred from buggy methods only?* By default, INVALIDATOR considers invariants inferred from all methods executed by a given test suite, rather than only using invariants inferred from buggy methods as done by prior works [59], [69]. In this research question, we investigate the effectiveness of 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 the *Overfitting-1* and *Overfitting-2* rules described in Section IV-A2 to identify overfitting patches. In this research question, we compare these two overfitting rules individually to evaluate their impact on INVALIDATOR's effectiveness.

C. Findings

1) RQ1: Effectiveness: We report the comparison of our approach, INVALIDATOR against baseline techniques consisting of RGT [74], ODS [71], BERT+LR [55], PATCHSIM [66], DIFFTGEN [65], ANTI-PATTERNS [53], GT-INVARIANT [69] on 139 APR-generated patches collected by Xiong et al. [66]. Table III presents the detailed results with respect to evaluation metrics given in Section V-A2, including *Recall*, *Precision*, *Accuracy*, and *F1-score*. We highlight the best result for each evaluation metric 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, equivalent to scores of 0.79, 0.97, 0.81, and 0.87 in terms of *Recall*, *Precision*, *Accuracy*, and *F1-score*, respectively. This implies that INVALIDATOR outperforms all baselines in *Recall*, *Accuracy*, and *F1-score* and obtains a good *Precision* of 0.97. We present more details below.

Accuracy. Table III shows that INVALIDATOR correctly identifies 86 out of 109 overfitting patches and 27 out of 30 correct patches, resulting in an *Accuracy* of 0.81. This indicates that INVALIDATOR outperforms the best baselines (ODS and RGT) by 19% and shows improvements of 23% to 146% compared to the other baselines.

F1-Score

F1-Score. INVALIDATOR outperforms the two best baselines (i.e., ODS and RGT) by 14%. Specifically, INVALIDATOR outperforms BERT+LR, PATCHSIM, DIFFTGEN, ANTI-PATTERNS, and GT-INVARIANT by 54%, 20%, 239%, 120%, and 29%, respectively. This is mainly because INVALIDATOR successfully identifies 79% of overfitting patches, whereas the best baselines filter out only 64% of overfitting patches while still maintaining an acceptable precision of 0.97.

Recall. With respect to *Recall*, INVALIDATOR achieves improvements of 23% (0.79 versus 0.64) compared to the best

TABLE III
COMPARISON OF THE EFFECTIVENESS OF INVALIDATOR WITH THE STATE-OF-THE-ART TECHNIQUES. THE BOLD NUMBERS DENOTES THE BEST RESULT FOR ACCURACY AND F1-SCORE

Techniques	TP	FN	FP	TN	Recall	Precision	Accuracy	F1-score
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**

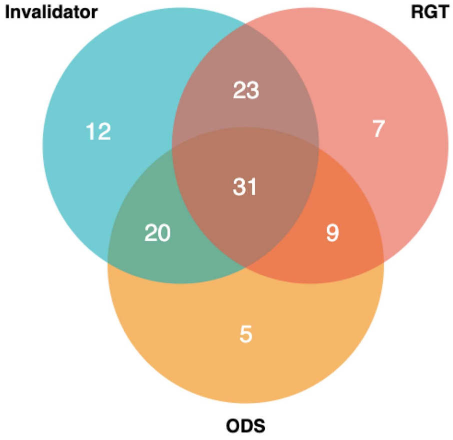
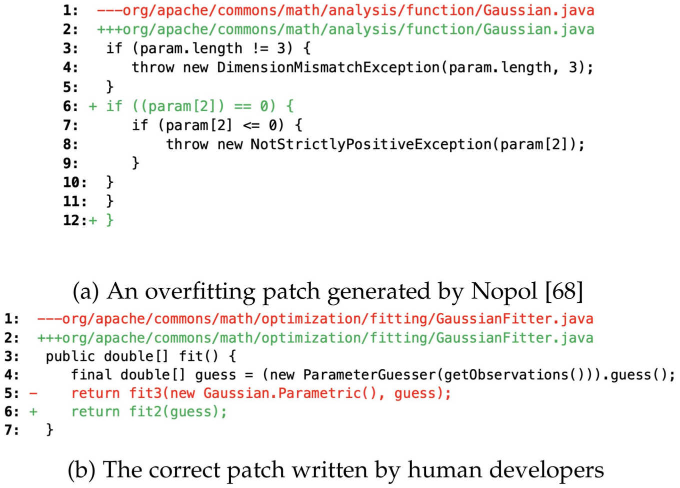


Fig. 5. Intersection on the correctly classifier overfitting patches by INVALIDATOR, ODS and RGT.

baselines (i.e., ODS and RGT). Specifically, INVALIDATOR outperforms RGT, ODS, BERT+LR, PATCHSIM, DIFFTGEN, ANTI-PATTERNS, and GT-INVARIANT by 23%, 23%, 100%, 39%, 438%, 219%, and 46%, respectively. This is mainly because INVALIDATOR leverages both syntactic and semantic reasoning while other techniques consider only syntax or semantics alone.

Precision. In terms of *Precision*, INVALIDATOR achieves a score of 0.97, outperforming ANTI-PATTERNS, ODS, RGT, and GT-INVARIANT, which have *Precision* scores of 0.96, 0.93, 0.93, and 0.89, respectively. However, INVALIDATOR slightly underperforms BERT+LR and PATCHSIM in terms of *Precision*. This may be because BERT+LR and PATCHSIM avoid filtering out correct patches by directly tuning the threshold of their classifier on the evaluation set, which could lead to overfitting on the set. In contrast, we tune our classification threshold on an independent validation set (as presented in Section V-A3) to avoid overfitting, which leads to lower precision than BERT+LR and PATCHSIM on the evaluation set. Meanwhile, DIFFTGEN has a perfect *Precision* (i.e., 1.0), but it is much less effective in filtering out overfitting patches, as reflected by its 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. Fig. 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 the 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.



```
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 [68]

```
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.

Case Study of Unique Overfitting Patches. To provide further insights into our approach, we manually analyzed unique overfitting patches that can be detected with the help of the novel techniques in INVALIDATOR. In Fig. 6, we present an example of an overfitting patch generated for the bug Math-58, which is detected as overfitting by INVALIDATOR but not RGT and ODS. In the bug, the `fit()` method (line 3 in Fig. 6(b)) is utilized to fit a Gaussian function to the observed points. Ideally, the method should ideally catch the exceptions of observed points having a negative standard deviation and return NaN values. To achieve 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. However, in the buggy version, `fit()` directly initializes a new Gaussian function and calls `fit3()` (line 5 in Fig. 6(b)), which results in the buggy version missing

TABLE IV
COMPARISON OF THE EFFECTIVENESS OF INVALIDATOR'S SYNTACTIC-BASED CLASSIFIER WITH THE STATE-OF-THE-ART TECHNIQUES. THE BOLD NUMBERS DENOTES THE BEST RESULT FOR ACCURACY, F1-SCORE AND AUC

Techniques	TP	FN	FP	TN	Recall	Precision	Accuracy	F1-score	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**

the observed points having a negative standard deviation and throwing NotStrictlyPositiveException. As we can see in Fig. 6(a), Nopol fixes the bug by adding the condition param[2] == 0 (line 6), which ensures that the NotStrictlyPositiveException (line 8) is unreachable when the observed points have a negative standard deviation. This leads to the failing test case being 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. In contrast, 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 III, in which an overfitting patch generated by Kali [47] cannot be detected by RGT, but can be detected by INVALIDATOR as the patch violates our Overfitting-1 rule, i.e., it violates correct behavior.

> *Answers to RQ1:* INVALIDATOR yields very promising performance on assessing the correctness of APR-generated patches (*Accuracy* at 0.81 and *F1-score* at 0.87) and outperforms the best baseline by 19% and 14% in terms of *Accuracy* and *F1-score*, respectively. Besides, the complementary use of the three best-performing techniques can cover 107/109 overfitting patches.

2) RQ2: Effectiveness of Syntactic-Based Classifier: [RQ2.1: Our syntactic-based classifier versus 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 IV presents the effectiveness of our approach and two baselines on six evaluation metrics including *Accuracy*, *F1-score*, and *AUC*. The experimental results demonstrate that INVALIDATOR significantly outperforms two baselines over six evaluation metrics. Particularly, INVALIDATOR yields an *Accuracy* of 0.73 and *F1-score* 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-score*, 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 versus 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 V 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-score*, and *AUC*, respectively. Meanwhile, the improvements without ground truth knowledge are 5%, 7%, and 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-score*, and *AUC*. The improvement is especially substantial regarding threshold-dependent techniques, i.e., *Accuracy* and *F1-score*. These results indicate the advantage of adding ground truth knowledge for syntactic-based classifiers.

TABLE V
Comparison of the Effectiveness of CodeBERT Features With ODS and BERT Features. The Bold Numbers Denotes the Best Result for Accuracy. F1-Score and AUC

ground truth	Techniques	TP	FN	FP	TN	Recall	Precision	Accuracy	F1-score	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**

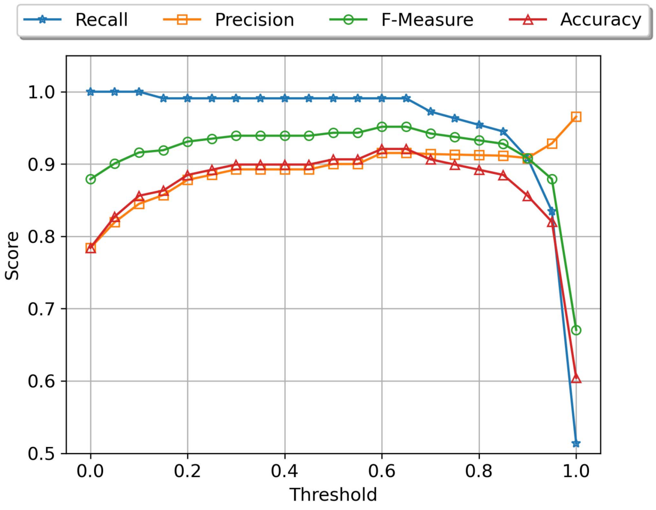


Fig. 7. The performance of Invalidator with different classification thresholds on the evaluation set.

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.

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 IV-B4. In this sub-question, we investigate the performance of Invalidator in terms of *Recall*, *Precision*, *F1-score*, and *Accuracy* with different classification thresholds in range (0, 1). The impact of the classification threshold on the performance of our approach is illustrated in Fig. 7.

We can see that the *Recall* holds steady at around *1.0* when the classification thresholds are in the range of (0.0, 0.65), then slightly decreases to about 0.94 (at the threshold of 0.85) before dropping to about *0.53* at the 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 at most of the thresholds. These results indicate that the assessment of Invalidator is reliable.

With respect to *Accuracy* and *F1-score*, the performance of Invalidator shares a similar trend on these metrics according to the variation of the classification threshold. In detail, *Accuracy* and *F1-score* 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 *F1-score* at the threshold of 0.9 before dropping to below 0.7 at the maximum threshold of 1.0.

In summary, our results suggest that the classification threshold has a limited impact on the *F1-score* and *Accuracy*, despite its influence on *Recall* and *Precision*. Practitioners and researchers can therefore select a threshold that aligns with their needs, without compromising the discriminative ability of APAC techniques, as reflected by *F1-score* and *Accuracy*.

Answers to RQ3.1: Despite the change of *Precision* and *Recall*, Invalidator still achieves promising overall performance, i.e., *F1-score* 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 *versus Existing threshold-dependence techniques]*

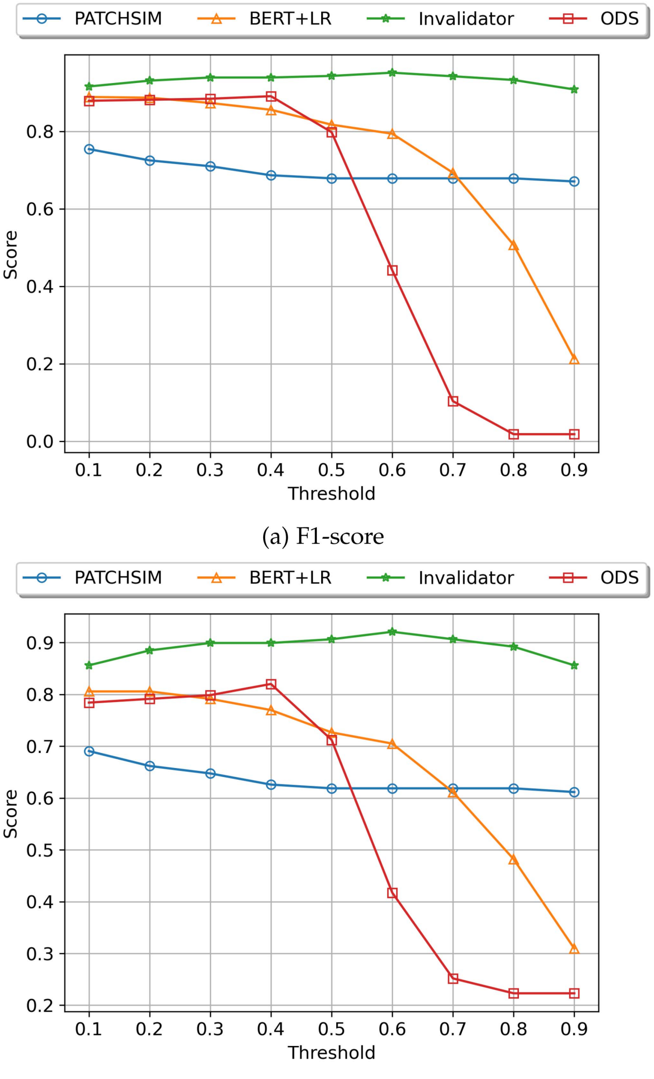
In this sub-question, we compare the performance, reflected by *F1-score* and *Accuracy)*, of four threshold-dependence techniques consisting of Invalidator, ODS [71], Bert+LR [55] and PatchSim [66] with nine different thresholds in the range of (0.1, 0.9). The impact of the classification threshold on the performance of threshold-dependence techniques is illustrated in Fig. 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 the threshold range of (0.1, 0.4) before witnessing a significant decrease of both *F1-score* 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 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.

TABLE VI
ABLATION STUDY. THE INVALIDATOR$_{Syn}$ AND INVALIDATOR$_{Sem}$ DENOTES INVALIDATOR'S SYNTACTIC AND SEMANTIC CLASSIFIERS, RESPECTIVELY

Techniques	TP	FN	FP	TN	Recall	Precision	Accuracy	F1-score
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

The bold numbers denotes the better result in each evaluation metric.



(a) F1-score

(b) Accuracy

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

TABLE VII
OVERALL PERFORMANCE OF INVALIDATOR'S *SEMANTIC CLASSIFIER* WITH DIFFERENT INVARIANT GRANULARITY: BUGGY METHODS AND EXECUTED METHODS

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

The bold numbers denotes the better result in each evaluation metric.

4) Ablation Study: [RQ4.1: The impact of semantic-based and syntactic-based classifiers 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 VI shows the results of our experiments. INVALIDATOR$_{sem}$, INVALIDATOR$_{syn}$ refer to semantic and syntactic-based classifiers, 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-score; 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 classifiers. 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 classifiers 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 VII, the invariants inferred from executed methods can boost the performance of our semantic classifier in APAC by 28% at *Accuracy* and 31% at *F1-score*. The key reason for the improvement is that behavioral

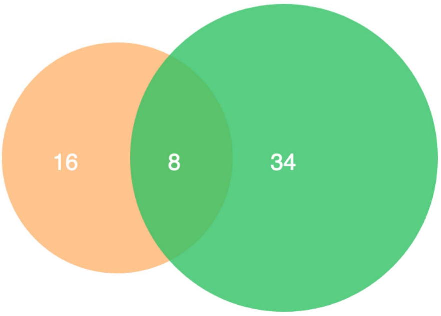


Fig. 9. The impact of overfitting rules on the performance of INVALIDATOR*'s semantic classifier*.

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 *F1-score*

[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 Fig. 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*.

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

VI. DISCUSSION

A. 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, 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 the assessment time of INVALIDATOR is reasonable but the invariant inference is time-consuming. However, the 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 a limited budget. However, even in the worst case, the performance of INVALIDATOR will only drop to the performance of our syntactic classifier, which still outperforms the state-of-the-art baselines. We leave the improvement on time efficiency of invariant inference for future work.

B. Potential Application

Although the reliance on ground truth patches limits our applications on pure APR problem settings, INVALIDATOR may be not only useful in patch correctness assessment but also in APR on problem settings where ground truth programs are available. For example, in the context of regression bug fixing [22], [52], a potential ground truth could be the original version before applying a bug-inducing commit. Besides, automated patch correctness assessment with ground truth cannot be directly used in automated program repair, it has been shown to be helpful in the training phase of learning-based program repair, in which the ground truth patches are available [75].

C. 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 and 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 correctness assessment in the APR community [18], [33], [59], [66]. Another threat to external validity is that patches in our dataset are only generated for the Defects4J dataset. This may not represent all bugs in real-world projects and thus may affect the generalizability of our findings. Unfortunately, besides 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 [18]. 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 in our study is that the correctness of the patches may be subject to subjective bias because they were manually labeled by human annotators, as mentioned in Section II-A. To mitigate this risk, we collected classification results from reliable sources that are widely used in the research community.

VII. RELATED WORK

A. Automated Program Repair

Our study investigates patches generated by several popular APR techniques, including GENPROG [27], KALI [48], NOPOL [68], HDREPAIR [23] and ACS [67]. GENPROG and KALI are heuristic-based techniques that construct a search space by using mutation operations and 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 [63], SIMFIX [13], FIXMINER [16], and TBAR [31] have been proposed to fix bugs automatically based on frequent fix patterns. Other approaches (e.g., SEQUENCER [3], DLFIX [28], COCONUT [36]) propose to generate patches by using deep learning models.

B. 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 [51]. Many APR techniques, e.g., GENPROG [27], RSREPAIR [47], AE [61], and ANGELIX [40] have been shown to suffer from the overfitting issue [24], [48].

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 [41]. Le et al. also suggested making publicly available to the community authors' evaluation on patch correctness [18]. 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 [18], or (2) manual inspection to compare APR-generated patches with the ground truth [16], [30], [31], [63]. Le et al. show that automated validation via an independent test suite is less effective than manual validation, but there is a potential risk of human bias when using manual validation [18]. Also, manual validation requires repetitive and expensive tasks, which automated validation can complement.

In this work, we use a data set of 885 APR-generated patches for large real-world programs whose correctness labels have been released by recent popular work [18], [33], [39], [59], [66]. 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.

C. 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.

1) Semantic-Based APAC: With respect to semantic-based APAC, the closely related works to our work are DIFFTGEN [65] and RGT [74] Similar to our work INVALIDATOR, the techniques identify 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 an 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 classifier 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. [76] 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 the majority of the studied plausible patches (92/96) expose different modifications of runtime behaviors captured by the program invariants, compared to correct patches [69]. However, this work does not propose any techniques to validate APR-generated patches. Based on the findings of Yang and Yang, Ye et al. [72], and Wang et al. [59] 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 [72]. Meanwhile, INVALIDATOR identifies patch correctness based on carefully designed overfitting behaviors by comparing invariants inferred from both buggy programs and developer-patched programs 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. [70] 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. [66] 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 of execution traces. Finally, it uses an enhanced test suite to determine whether an 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.

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. [55]. BERT+LR assumes that correct codes differ substantially from incorrect codes and uses code representation techniques to differentiate between them. Specifically, BERT+LR embeds a patched code and a buggy code into numerical vectors using BERT [5] and then uses Logistic Regression to estimate the similarity between them. Finally, a patch is considered incorrect/overfitting if the similarity is lower than a certain threshold. However, determining a suitable threshold is challenging because the difference between correct and incorrect codes can vary among programs. In contrast, our approach considers the similarity of a patched program to its ground truth and buggy program. 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. Thus, the similarity between a patched and ground truth code serves as a “soft threshold” that can be adjusted for different programs. As a result, our approach is more flexible than BERT+LR and achieves better performance, as demonstrated in Section V-C. Additionally, our syntactic-based classifier incorporates new syntactic features from CodeBERT [8], which has been shown to be more effective than BERT features.

Other works rely on hand-crafted code features to validate the generated patch, including ANTI-PATTERNS and ODS. Tan et al. [53] propose anti-patterns (i.e., specific static structures) to filter out overfitting patches. Ye et al. [71] 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 pre-trained 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.

VIII. CONCLUSION

In this paper, 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* specifications of a program under test. Based on the inferred specifications, INVALIDATOR effectively identifies whether an 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 pre-trained model of code, to represent the language semantics of program syntax via a vector of numbers and then measures syntactic differences between APR-generated patches and their buggy and correct versions. Based on syntactic differences, INVALIDATOR uses a trained model from labeled patches to estimate the likelihood of an APR-generated patch being overfitting. We compared INVALIDATOR against state-of-the-art automated patch correctness assessment techniques from a popular dataset 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 ground truths which are available (e.g., such as the original version of the 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 the training process to further improve learning-based program repair, as inspired by RewardRepair [75].

IX. DATA AVAILABILITY

INVALIDATOR is publicly available at https://github.com/thanhlecongg/Invalidator. All materials including implementation, datasets, and experimental results are also published via https://doi.org/10.5281/zenodo.7699142

ACKNOWLEDGMENT

We would like to thank Prof. Domenico Bianculli and the anonymous reviewers for their insightful feedback and comments.

REFERENCES

[1] T.-D. B. Le, D. Lo, C. L. Goues, and L. Grunske, “A learning-to-rank based fault localization approach using likely invariants,” in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 177–188.

[2] T.-D. B. Le, D. Lo, C. L. Goues, and L. Grunske, “A learning-to-rank based fault localization approach using likely invariants,” in *Proc. ACM 25th/26th Int. Symp. Softw. Testing Anal.*, New York, NY, USA, 2016, pp. 177–188.

[3] 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 Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1943–1959, Sep. 2021.

[4] L. D. Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, Springer, 2008, pp. 337–340.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2018, pp. 4171–4186.

[6] Z. Y. Ding, Y. Lyu, C. Timperley, and C. L. Goues, “Leveraging program invariants to promote population diversity in search-based automatic program repair,” in *Proc. IEEE Int. Workshop Genet. Improvement*, 2019, pp. 2–9.

[7] M. D. Ernst et al., “The daikon system for dynamic detection of likely invariants,” *Sci. Comput. Program.*, vol. 69, no. 1/3, pp. 35–45, 2007.

[8] Z. Feng et al., “CodeBERT: A pre-trained model for programming and natural languages,” in *Proc. ACM Conf. Empir. Methods Natural Lang. Process.*, 2020, pp. 1536–1547.

[9] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 416–419.

[10] D. Gopinath, H. Converse, C. Pasareanu, and A. Taly, "Property inference for deep neural networks," in *Proc. IEEE/ACM 34th Int. Conf. Automated Softw. Eng.*, 2019, pp. 797–809.

[11] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "CC2Vec: Distributed representations of code changes," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 518–529.

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

[13] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 298–309.

[14] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. 23th Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440.

[15] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. IEEE 35th Int. Conf. Softw. Eng.*, 2013, pp. 802–811.

[16] A. Koyuncu et al., "FixMiner: Mining relevant fix patterns for automated program repair," *Empirical Softw. Eng.*, vol. 25, pp. 1–45, 2020.

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

[18] D. X. B. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *Proc. IEEE 41st Int. Conf. Softw. Eng.*, 2019, pp. 524–535.

[19] T. C. Le, T. Antonopoulos, P. Fathololumi, E. Koskinen, and T. Nguyen, "Dynamite: Dynamic termination and non-termination proofs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[20] X.-B. D. Le, D.-H. Chu, D. Lo, C. L. Goues, and W. Visser, "JFIX: Semantics-based repair of Java programs via symbolic pathfinder," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 376–379.

[21] X.-B. D. Le, D.-H. Chu, D. Lo, C. L. Goues, and W. Visser, "S3: Syntax-and semantic-guided repair synthesis via programming by examples," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 593–604.

[22] X.-B. D. Le and Q. L. Le, "ReFixar: Multi-version reasoning for automated repair of regression errors," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng.*, 2021, pp. 162–172.

[23] X. B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, 2016, pp. 213–224.

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

[25] T. Le-Cong et al., "Autopruner: Transformer-based call graph pruning," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 520–532.

[26] T. Le-Cong, X. B. D. Le, Q. T. Huynh, and P. L. Nguyen, "Usability and aesthetics: Better together for automated repair of web pages," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng.*, 2021, pp. 173–183.

[27] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.

[28] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: Context-based code transformation learning for automated program repair," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 602–614.

[29] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, pp. 1–29, 2022.

[30] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: Fixing semantic bugs with fix patterns of static analysis violations," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reengineering*, 2019, pp. 1–12.

[31] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBAR: Revisiting template-based automated program repair," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 31–42.

[32] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, "LSRepair: Live search of fix ingredients for automated program repair," in *Proc. IEEE 25th Asia-Pacific Softw. Eng. Conf.*, 2018, pp. 658–662.

[33] K. Liu et al., "On the efficiency of test suite based program repair," in *Proc. Int. Conf. Softw. Eng.*, 2020, pp. 615–627.

[34] 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 *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2009, pp. 557–566.

[35] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2016, pp. 298–312.

[36] 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 *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 101–114.

[37] S. Mahajan, N. Abolhassani, P. McMinn, and W. G. J. Halfond, "Automated repair of mobile friendly problems in web pages," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 140–150.

[38] A. Marginean et al., "SapFix: Automated end-to-end repair at scale," in *Proc. IEEE 41st Int. Conf. Softw. Eng., Softw. Eng. Pract.*, 2019, pp. 269–278.

[39] 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 Softw. Eng.*, vol. 22, no. 4, pp. 1936–1964, 2017.

[40] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 691–701.

[41] 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 *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 234–242.

[42] 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 *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. New Ideas Emerg. Results*, 2022, pp. 116–120.

[43] T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks, "Counterexample-guided approach to finding numerical invariants," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 605–615.

[44] T. G. Nguyen, T. Le-Cong, H. J. Kang, X.-B. D. Le, and D. Lo, "VulCurator: A vulnerability-fixing commit detector," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 1726–1730.

[45] A. Nilizadeh, G. T. Leavens, X.-B. D. Le, C. S. Păsăreanu, and D. R. Cok, "Exploring true test overfitting in dynamic automated program repair using formal methods," in *Proc. IEEE 14th Conf. Softw. Testing Verification Validation*, 2021, pp. 229–240.

[46] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Proc. 22nd ACM SIGPLAN Conf. Object-oriented Program. Syst. Appl. Companion*, 2007, pp. 815–816.

[47] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 254–265.

[48] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36.

[49] P. Sagdeo, N. Ewalt, D. Pal, and S. Vasudevan, "Using automatically generated invariants for regression testing and bug localization," in *Proc. IEEE/ACM 28th Int. Conf. Automated Softw. Eng.*, 2013, pp. 634–639.

[50] 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 *Proc. IEEE/ACM 30th Int. Conf. Automated Softw. Eng.*, 2015, pp. 201–211.

[51] E. K. Smith, E. T. Barr, C. L. Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 532–543.

[52] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proc. IEEE 37th Int. Conf. Softw. Eng.*, 2015, pp. 471–482.

[53] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 727–738.

[54] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: A human study," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 64–74.

[55] H. Tian et al., "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 981–992.

[56] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *Proc. IEEE 34th Int. Conf. Softw. Eng.*, 2012, pp. 386–396.

[57] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *Proc. IEEE 19th Work. Conf. Reverse Eng.*, 2012, pp. 215–224.

[58] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[59] S. Wang et al., "Automated patch correctness assessment: How far are we?," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 968–980.

[60] S. Wang and J. Jiang, "A compare-aggregate model for matching text sequences," in *Proc. Int. Conf. Learn. Representations*, Toulon, France, Apr. 2017, pp. 1–15.

[61] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proc. IEEE/ACM 28th Int. Conf. Automated Softw. Eng.*, 2013, pp. 356–366.

[62] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 364–374.

[63] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 1–11.

[64] C. Steven Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 959–971.

[65] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proc. 6th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 226–236.

[66] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 789–799.

[67] Y. Xiong et al., "Precise condition synthesis for program repair," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 416–426.

[68] J. Xuan et al., "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017.

[69] B. Yang and J. Yang, "Exploring the differences between plausible and correct patches at fine-grained level," in *Proc. IEEE 2nd Int. Workshop Intell. Bug Fixing*, 2020, pp. 1–8.

[70] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 831–841.

[71] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 2920–2938, Aug. 2022.

[72] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," in *Proc. IEEE 1st Int. Workshop Intell. Bug Fixing*, 2019, pp. 1–10.

[73] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Self-APR: Self-supervised program repair with test execution diagnostics," 2022, *arXiv:2203.12755*.

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

[75] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1506–1518.

[76] 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 Softw. Eng.*, vol. 24,. no. 1, pp. 33–67, 2019.

[77] J. Zhou et al., "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *Proc. IEEE/ACM 36th Int. Conf. Automated Softw. Eng.*, 2021, pp. 705–716.

[78] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codeBERT," in *Proc. Int. Conf. Softw. Maintenance Evol.*, 2021, pp. 425–436.

[79] X. Zhou et al., "PatchZero: Zero-shot automatic patch correctness assessment," 2023, *arXiv:2303.00202*.

Thanh Le-Cong (Graduate Student Member, IEEE) received the bachelor's degree in information technology from the Hanoi University of Science and Technology, Vietnam. He is currently working toward the PhD degree with the School of Computing and Information Systems, The University of Melbourne in Australia. His research interest lies on the intersection between artificial intelligence and software engineering, with a current focus on automated program repair and program analysis.

Duc-Minh Luong received the bachelor's degree in information technology from the Hanoi University of Science and Technology, Vietnam. He is currently a software engineer with Sun Aterisk Inc, Japan. His research interests include machine learning applied to software debugging and automated program repair.

Xuan Bach D. Le received the PhD degree from Singapore Management University, in 2018. He is currently a lecturer with The University of Melbourne Australia. Before that, he was a postdoctoral researcher with CyLab, Carnegie Mellon University. His research interests span software engineering and programming languages, including: software mining, empirical software engineering, program analysis, repair, synthesis, and verification.

David Lo (Fellow, IEEE) is a professor and director with the Information Systems and Technology Cluster, School of Computing and Information Systems, Singapore Management University. His research interest lies in the intersection of software engineering, cybersecurity and data science, encompassing socio-technical aspects and analysis of different kinds of software artefacts, with the goal of improving software quality and security and developer productivity. He has won more than 20 international research and service awards including 6 ACM SIGSOFT Distinguished Paper Awards, 2 Most Influential Paper (or Test-of-Time) awards, and the 2021 IEEE TCSE Distinguished Service Award. He has served in more than 40 organizing committees, including serving as a general/program co-chair of ICSE 2025, ESEC/FSE 2024, MSR 2022, ASE 2020, SANER 2019, ICSME 2018, etc. He is also serving on the editorial boards of a number of journals including *IEEE Transactions on Software Engineering*, *Empirical Software Engineering*, *IEEE Transactions of Reliability*, *Communications of the ACM*, and *ACM Computing Survey*. He is an ASE Fellow (2021), and ACM distinguished member (2019).

Nhat-Hoa Tran received the BS and MS degrees in information technology from the Hanoi University of Science and Technology, and the PhD degree from the Japan Institute of Science and Technology (JAIST). He is a lecturer with the School of Information and Communication Technololy, Hanoi University of Science and Technology, Vietnam. His primary research interests are software engineering, artificial intelligence, and formal method.

Quang-Huy Bui received the bachelor's and master's degrees in information technology from the Hanoi University of Science and Technology, Vietnam. He is currently a software engineer with FPT Software Co. Ltd., Vietnam. His research interests include software development and testing.

Quyet-Thang Huynh received the PhD degree in information and computer sciences from the Varna Technical University, Bulgaria. He is an associate professor and president of the Hanoi University of Science and Technology, Vietnam. His primary research interests are software quality, testing and project management.