Enhancing APR with PRISM: A Semantic-Based Approach to Overfitting Patch Detection

DOWON SONG, Korea University, Republic of Korea
HAKJOO OH*, Korea University, Republic of Korea

We present Prism, a novel technique for detecting overfitting patches in automatic program repair (APR). Despite significant advances in APR, overfitting patches—those passing test suites but not fixing bugs—persist, degrading performance and increasing developer review burden. To mitigate overfitting, various automatic patch correctness classification (APCC) techniques have been proposed. However, while accurate, existing APCC methods often mislabel scarce correct patches as incorrect, significantly lowering the APR fix rate. To address this, we propose (1) novel semantic features capturing patch-induced behavioral changes and (2) a tailored learning algorithm that preserves correct patches while filtering incorrect ones. Experiments on ranked patch data from 10 APR tools show that Prism uniquely reduces review burden and finds more correct patches. Other methods lower the fix rate by misclassifying correct patches. Evaluations on 1,829 labeled patches confirm Prism removes more incorrect patches at equal correct patch preservation rates.

CCS Concepts: • **Software and its engineering → Software verification and validation**.

Additional Key Words and Phrases: Automated Program Repair, Static Analysis

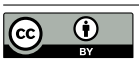
ACM Reference Format:
Dowon Song and Hakjoo Oh. 2025. Enhancing APR with PRISM: A Semantic-Based Approach to Overfitting Patch Detection. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 392 (October 2025), 29 pages. https://doi.org/10.1145/3763170

1 Introduction

Overfitting Patches in APR. Despite the remarkable progress in automatic program repair (APR) over the past decade [1, 17, 19, 20, 23, 25, 29, 30, 33–36, 40, 48, 49, 60], modern APR tools still suffer from a major challenge called overfitting. This problem occurs when APR techniques generate patches that only satisfy the given test suite as a specification, but fail to generalize to the full intended program behavior. Overfitting leads to the creation of incorrect patches, which satisfy the provided tests without actually fixing the underlying bug. As a result, it undermines not only the overall performance but also the reliability of APR systems [18, 21, 39, 41]. For example, if an APR technique returns only the first plausible patch that satisfies the given tests, overfitting can cause the APR to produce an incorrect patch that prevents the discovery of the correct solution. Alternatively, if APR tools produce a set of plausible patches like a recommendation system, overfitting may result in the generation of numerous incorrect patches, thereby increasing the review effort required by developers to identify the truly correct patch.

*Corresponding author

Authors' Contact Information: Dowon Song, Korea University, Republic of Korea, dowon_song@korea.ac.kr; Hakjoo Oh, Korea University, Republic of Korea, hakjoo_oh@korea.ac.kr.



ACM 2475-1421/2025/10-ART392
https://doi.org/10.1145/3763170

Table 1. Performance of TBar with different APCC techniques. "# Bugs with Top-k Fixes" (Top-1, Top-5, Top-∞) measures the APR effectiveness. "# Patches to Review" (To 1st Fix, To All Fixes) measures the number of patches a developer needs to examine before finding the first or all correct patches.

Settings	# Bugs with Top-k Fixes (△%)			# Patches to Review (△%)	
	Top-1	Top-5	Top-∞	To 1st Fix	To All Fixes
TBar (Baseline)	29	33	37	142	219
TBar + PATCH-SIM	22 (-24%)	26 (-21%)	28 (-24%)	84 (-41%)	153 (-30%)
TBar + ODS	13 (-55%)	14 (-58%)	15 (-59%)	**29 (-80%)**	**29 (-87%)**
TBar + Shibboleth	25 (-14%)	29 (-12%)	37 (-)	126 (-11%)	275 (+26%)
TBar + **Prism**	**32 (+10%)**	**34 (+3%)**	37 (-)	96 (-32%)	166 (-24%)

Automatic Patch Correctness Classification (APCC). To combat the overfitting problem, numerous APCC (Automatic Patch Correctness Classification) techniques have been proposed [7, 10, 13, 24, 43, 45, 46, 52, 55, 56, 61]. These approaches aim to identify and filter out incorrect patches so that APR systems can focus on generating or selecting patches that are more likely to be correct. Existing methods can be broadly divided into two categories. On one hand, dynamic execution-based techniques [13, 52, 55] leverage runtime information, such as memory-safety checks, crash oracles [13, 55], or execution traces [52]. These methods detect incorrect patches by identifying deviations from expected runtime behavior. On the other hand, syntax-based approaches [24, 43, 45, 46, 56, 61] classify patch correctness based on code structure and patch patterns. Many of these studies incorporate machine learning techniques with human-crafted code features [43, 56] or automatically extracted features [24, 45, 46]. In addition, the use of large language models (LLMs) has recently been considered [61], as they require minimal fine-tuning and can leverage massive datasets. Furthermore, some methods, such as Shibboleth [10], combine both dynamic and syntactic information to achieve a balanced classification. Prior works have demonstrated promising performance on widely used APR-generated patch datasets [27, 47, 52, 58]; for example, Tian et al. [46] showed that integrating human-crafted syntactic features [56] with learned embeddings [45] can classify 2,147 patches with an accuracy of 76%.

Limitation of Existing APCC Techniques. Despite their impressive classification accuracies, existing APCCs have a critical limitation when integrated with APR systems. While these approaches can reduce the human effort in patch review by aggressively filtering out incorrect patches, they are likely to conversely impair the most important performance metric (i.e., fix rate) of APR. Table 1 shows the performance change of an APR system, TBar, when combined with representative APCC techniques. We observed that all of the existing APCC techniques degrade TBar's performance relative to the baseline. This counterintuitive result arises from the unique characteristics of APR: correct patches are extremely scarce in the search space relative to incorrect ones [39]. Thus, in a conventional binary classification setting, labeling most patches as incorrect might boost overall accuracy. However, in the actual APR context, even a single misclassified correct patch can drastically reduce the fix rate [13]. *Therefore, an effective APCC technique in practice must preserve almost all correct patches while filtering out only the incorrect ones.*

Our Approach. In this paper, we present Prism, a novel APCC technique that uniquely achieves both enhanced APR performance and reduced human effort in patch review. This achievement is underpinned by two key components: (1) a patch representation based on *semantic features* that directly captures the behavioral impact induced by patches, and (2) a *specialized learning algorithm* to obtain a patch classification model tailored for APR.

To achieve our first objective, instead of relying on changes in test executions or syntactic differences, which only provide indirect indications of a patch's behavioral impact, we introduce 66 new semantic features that explicitly capture the modifications caused by a patch. These features are defined based on 11 general semantic properties. In contrast to test execution-based methods, which can only cover a small set of a program's executions, we employ static analysis that effectively approximates a larger set of possible program behaviors. Specifically, we develop a differential static analysis to identify differences in semantic properties between the buggy and patched versions. These differences are then used to extract semantic features from the given patch.

Second, Prism employs a custom learning algorithm specifically designed for the APR context. Our algorithm produces a boolean formula over the semantic features to express accurate descriptions of the correct patches. Since it is crucial to preserve correct patches, which are extremely rare in the APR search space, our algorithm ensures a high preservation rate of correct patches while maximizing the filtering of incorrect ones on the training set. Furthermore, unlike existing off-the-shelf learning algorithms, our model is inherently interpretable due to its boolean formula format, providing clear and human-understandable reasoning behind each classification decision.

We conducted large-scale experiments to demonstrate that Prism is the only APCC technique that benefits APR in real-world scenarios. In experiments using ranked patch data from 10 APR tools (Section 4.1), Prism reduced human effort in reviewing top-1 patches by 12% and identified 9 more top-1 correct patches compared to the baseline. In contrast, although existing techniques [10, 52, 56] reduced review effort by 9% to 84%, they mistakenly filtered out 24 to 118 top-1 correct patches, drastically lowering the fix rate. Furthermore, experiments on 1,829 labeled patches (Section 4.2) show that, at the same correct patch preservation rate, Prism consistently detects and filters out more incorrect patches. Our ablation study confirms that both components of Prism are essential (Section 4.3). Lastly, the analysis of our experimental results shows that Prism requires minimal overhead, making it practical for real-world APR applications while providing interpretable insights that offer valuable intuition for further refinements in APR (Section 4.4).

Contributions. This paper makes the following contributions:

- We propose Prism, the only APCC technique that reduces human burden while improving the fix rates of APR tools.
- We introduce 66 semantic features that directly express the behavioral impact of patches, extracted using a specialized differential static analysis.
- We develop a learning algorithm tailored for APR that generates an effective patch classifier.
- We conduct large-scale experiments to validate that Prism outperforms existing techniques.
- For open science, we make Prism open-sourced and the datasets publicly available.

2 Overview

In this section, we illustrate the primary challenge in accurately classifying patch correctness with current techniques and demonstrate how Prism addresses it. Figure 1 shows two patches from the Defects4J benchmark:

(a) A correct patch for the Chart-1 bug, generated by the APR tool TBar [25].
(b) An incorrect patch for the Chart-25 bug, produced by the APR tool jMutRepair[32].

Notably, the patch in Figure 1a is the *only correct fix* among those generated by TBar. Therefore, misclassifying it as incorrect would prevent TBar from successfully repairing Chart-1. Our primary objective is to correctly classify both patches, with special attention to avoiding misclassification of the correct one, as such an error severely impairs the repair performance of APR tools.

```
1 public Collection getLegendItems() {
2    int idx = this.plot.getIdx(this);
3    DSet dataset = this.plot.getDSet(idx);
4 -  if (dataset != null) {
5 +  if (dataset == null) {
6       return result;
7    }
8    int seriesCnt = dataset.getRowCount();
9    ...}
```

(a) The only correct patch generated by TBar for the Chart-1 bug

```
1 public Number getMean(int r, int c) {
2    Number result = null;
3    MASD masd = this.data.getObj(r, c);
4 -  if (masd != null) {
5 +  if (masd == null) {
6       result = masd.getMean();
7    }
8    return result;
9 }
```

(b) An incorrect patch generated by jMutRepair for the Chart-25 bug

Fig. 1. Motivating examples with two patches

Limitations of Existing Techniques. We found that existing techniques [10, 52, 56] are unable to distinguish between the two patches, leading to the misclassification of the only correct patch in Figure 1a and, consequently, a reduction in TBar's repair performance.

ODS [56], a syntax-based probabilistic approach, fails to distinguish between the two example patches. This technique assumes that patches with similar syntactic features have the same correctness. However, in this example, the two patches are syntactically similar because they both modify null-handling conditions for local variables (`dataset` and `masd`), yet they differ in correctness. As a result, ODS labels both patches as incorrect, ultimately degrading the fix rate of TBar.

Similarly, PATCH-SIM [52], a test execution-based method, misclassifies the correct patch. It operates under the assumption that patches causing significant changes in passing test traces are likely to be incorrect, while those altering failing test traces are considered correct. However, the correct patch in Figure 1a contradicts this assumption. It significantly changes the execution trace of passing tests by switching a conditional branch from `true` to `false`.

Even Shibboleth [10], which integrates both syntactic and execution features, misclassifies the correct patch. It presumes that a correct patch should introduce minimal modifications to syntax, execution traces, and the coverage of passing tests. However, the patch in Figure 1a makes only a slight syntactic modification while causing substantial changes in test coverage.

Our Technique: Prism. To distinguish correct from incorrect patches, we focus on the semantic changes each patch introduces. This is motivated by the fact that existing methods do not directly capture the behavioral impact of a patch, but only describe its underlying semantic modifications indirectly. For example, an error may cause abnormal termination; a trace-based approach might only show a shortened trace rather than the termination itself. In contrast, our approach directly captures and details the semantic changes introduced by a patch, offering a clear understanding of its behavioral impact. For the patches shown in Figure 1, we identified two critical semantic differences that clearly distinguish the correct patch from the incorrect one:

(1) *Patch Introduces a New Null Pointer Exception*: The patch in Figure 1b makes `masd` always `null` at line 7, causing an inevitable null pointer exception (NPE). In contrast, the patch in Figure 1a ensures `dataset` is non-null at line 9, thereby eliminating a potential NPE.
(2) *Patch Makes a Constant Method*: Before the patch, the method in Figure 1b returns different values depending on `masd`. After the patch, however, it consistently returns a null-initialized `result` (except when an NPE occurs). This change is not observed in Figure 1a.

First, to effectively express such semantic differences, we represent a patch with a novel set of 66 semantic features (Section 3.2). To extract semantic features from a patch, we employ a differential static analysis specifically tailored to patch changes. Unlike test execution, which only reveals a subset of runtime behaviors, our method considers all potential program executions. For instance, the memory states and corresponding path conditions at line 9 of the program before (left) and after (right) the patch in Figure 1a are obtained as follows:

Path Cond	Memory
α = `null`	$\bot$ (Unreachable By NPE)

Path Cond	Memory
$\alpha \neq$ `null`	`dataset` $\mapsto \alpha$ `dataset` $\mapsto \alpha$.`RowCount`

where α denotes an object that the variable `dataset` points to. In the analysis, before the patch, the object α is `null`. Consequently, when the code calls `dataset.getRowCount()` at line 9, an NPE is inevitably triggered. In contrast, after the patch, proper null-handling is introduced for `dataset`, α is non-null at line 9, thereby eliminating the potential for an NPE. Based on this difference, PRISM extracts the feature **Eliminate Null Pointer Exceptions**. Similarly, consider the analysis results at line 8 of the program before (left) and after (right) the patch in Figure 1b:

Path Cond	Memory
β = `null`	`masd` $\mapsto \beta$ `result` $\mapsto$ `null`
$\beta \neq$ `null`	`masd` $\mapsto \beta$ `result` $\mapsto \beta$.`mean`

Path Cond	Memory
β = `null`	$\bot$ (Unreachable By NPE)
$\beta \neq$ `null`	`masd` $\mapsto \beta$ `result` $\mapsto$ `null`

where β denotes an object referenced by the variable `masd`. According to our analysis, the program exhibits different behaviors depending on the value of β. When β is `null`, the pre-patch version of the function returns `null` without error. In contrast, after the patch, an NPE is triggered during the call to `masd.getMean()`. Furthermore, when β is non-null, the function originally returns the value of the `mean` field of `masd`. However, the post-patch function consistently returns `null`, behaving as a deterministic function. Consequently, these differences enable PRISM to extract two features: **Generate Null Pointer Exceptions** and **Generate Constant Methods**.

To classify the correctness of patches represented by semantic features, we employ a specialized learning algorithm. To improve the likelihood of identifying correct patches without degrading APR's overall performance, it is essential to maximize the retention of correct patches while discarding incorrect ones. To address this challenge, we have designed a customized learning algorithm that automatically derives a patch classification model. Our algorithm guarantees a specified correct patch preservation ratio while filtering out as many incorrect patches as possible on the given training set. Our learning algorithm returns a boolean formula in Disjunctive Normal Form (DNF) that characterizes correct patches. In our experiments, we found that the learned formula includes a key clause with 11 semantic features, which distinguishes the two patches:

Eliminate Null Pointer Exceptions $\wedge$ **Avoid Generating Constant Methods**
$\wedge$ Preserve Used Parameters $\wedge$ Eliminate Exceptional Calls $\wedge$ Preserve Constant Fields
$\wedge$ Do Not Eliminate Arrays with IOB $\wedge$ Preserve Return-Related Parameters
$\wedge$ Preserve Constant Returns $\wedge$ Avoid Generating Exceptional Calls
$\wedge$ Preserve Used Local Variables $\wedge$ Preserve Thrown Exceptions

Unlike off-the-shelf learning algorithms, our model is interpretable due to its boolean formula representation. From the learned formula, we found that it includes two key semantic features (i.e., Eliminate Null Pointer Exceptions and Avoid Generating Constant Methods) which play a critical role in distinguishing the two patches in Figure 1. As a result, while existing techniques fail

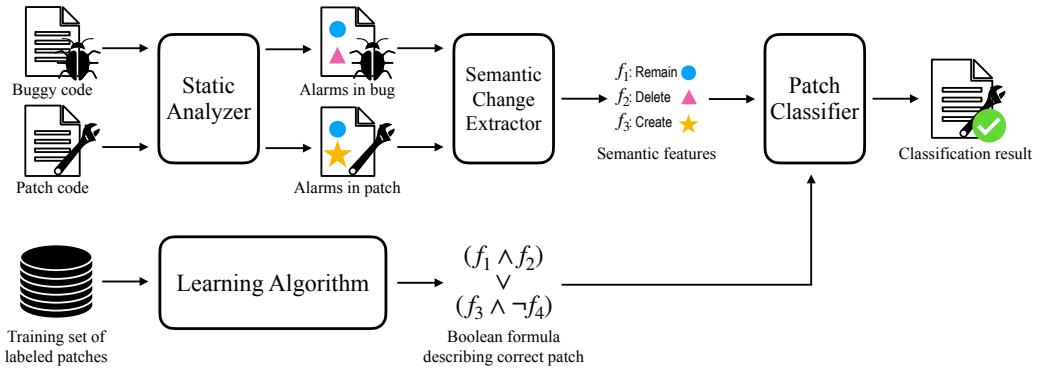


Fig. 2. Overall workflow of Prism

to classify the patches correctly or offer any explanation, our approach not only ensures accurate classification but also provides interpretable reasoning behind each decision.

3 Prism

In this section, we present how Prism works. Figure 2 illustrates its overall workflow. Given a pair of buggy and patched code, Prism first performs static analysis to capture the alarms of each version. It then compares the analysis results to extract semantic features that represent the behavioral impact of the patch. Finally, Prism learns a patch classifier from labeled data, producing a boolean formula that characterizes correct patches. The rest of this section is organized as follows. Section 3.1 formally defines the patch classification problem. Section 3.2 describes semantic features and how they are extracted using static analysis. Section 3.3 presents our learning algorithm.

3.1 Problem Definition

We begin by formally defining programs, patches, and the patch classification problem.

Program and Patch. To better illustrate the key mechanism of Prism, we assume a simple imperative style language represented by a control flow graph (CFG). For simplicity, we assume that loops and recursion are unrolled up to a fixed bound, resulting in a loop-free program. A program P consists of a collection of method declarations $MDecl_P$, a set of program points $\mathbb{L}_P$, a function $\mathsf{next}_P : \mathbb{L}_P \to \mathcal{P}(\mathbb{L}_P)$ for next labels, and a corresponding statement of each label $\mathsf{stmt}_P : \mathbb{L}_P \to \mathbb{S}$. Each method declaration $M \in MDecl_P$ is defined by a method name $m \in Mthd_P$, a parameter $p_m \in Var_P$, a return variable $ret_m \in Var_P$, a start label $\ell^s_m \in \mathbb{L}_P$, and an exit label $\ell^e_m \in \mathbb{L}_P$. We omit the subscript m when the method context is clear. In summary, the program P is defined as follows:

$$\begin{array}{rcrcl} Pgm & \ni & P & = & \langle MDecl_P, \mathbb{L}_P, \mathsf{next}_P, \mathsf{stmt}_P \rangle \\ MDecl_P & \ni & M & = & \langle m, p_m, \ell^s_m, \ell^e_m, ret_m \rangle \end{array}$$

A statement $S \in \mathbb{S}$, an l-value lv, and an experssion E are defined as follows:

$$\begin{array}{rcl} S & \to & lv := E \mid lv := \texttt{new}\ C() \mid \texttt{new}\ x[y] \mid \texttt{throw new}\ C \mid \texttt{assume}(x \prec y) \mid \texttt{catch}\ C \\ lv & \to & x \mid x[y] \mid x.y \\ E & \to & lv \mid n \mid \texttt{null} \mid m(x) \end{array}$$

A statement S can be an assignment, an object creation, an array allocation, a throw statement with an exception of class $C \in Class$, a binary relation $\prec \in \{=, \neq, >, \geq\}$ between two variables, or a catch statement that handles exceptions of type C. An l-value is either a variable, an array element,

or an object with field access. An expression is an l-value lv, an integer $n \in \mathbb{Z}$, `null`, or a method invocation $m(x)$. Note that, for simplicity, we model method calls such as $x.m(y)$ as function calls $m(x)$. We denote the expression used in the statement at $\mathsf{stmt}_P(\ell)$ as $\mathsf{E}(\ell)$. The variables used and defined at label ℓ are denoted as $\mathsf{use}(\ell)$ and $\mathsf{def}(\ell)$, respectively. For presentation simplicity, we assume that variables and method names are unique, and introduce two special variables: x_{size} for the size of array x and exn_m for the exception state of method m. Note that this simplified language is used for clarity of presentation. Our actual implementation is based on Infer's intermediate representation (SIL) and supports full Java features.

We define a patch as a pair of programs: the original (buggy) program $P_b \in Pgm$ and its modified version $P_p \in Pgm$. We write $\mathbb{L}_{Patch} = \mathbb{L}_{P_b} \Delta \mathbb{L}_{P_p}$ for all patched labels. For the labels that are not modified by the patch ($\ell \in \mathbb{L}_{P_b} \cap \mathbb{L}_{P_p}$), the programs have the same statements: $\mathsf{stmt}_{P_b}(\ell) = \mathsf{stmt}_{P_p}(\ell)$.

Patch Representation. In our approach, patches are represented by boolean formulas over *atomic features*. An atomic feature $f_i \in Feature$ is a predicate on patches (pairs of programs) and checks whether a specific aspect is present ($\top$) or absent ($\bot$):

$$f_i : Pgm \times Pgm \to \{\top, \bot\}$$

A boolean formula ψ is defined over the atomic features as follows:

$$\psi \to \top \mid \bot \mid f_i \in Feature \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2$$

Given a patch (P_b, P_p), a boolean formula ψ evaluates to $\top$ iff the patch is described by ψ:

$$\begin{array}{rcl} [\![\top]\!](P_b, P_p) & = & \top \\ [\![\bot]\!](P_b, P_p) & = & \bot \\ [\![f_i]\!](P_b, P_p) & = & f_i(P_b, P_p) \\ [\![\neg\psi]\!](P_b, P_p) & = & \neg[\![\psi]\!](P_b, P_p) \\ [\![\psi_1 \wedge \psi_2]\!](P_b, P_p) & = & [\![\psi_1]\!](P_b, P_p) \wedge [\![\psi_2]\!](P_b, P_p) \\ [\![\psi_1 \vee \psi_2]\!](P_b, P_p) & = & [\![\psi_1]\!](P_b, P_p) \vee [\![\psi_2]\!](P_b, P_p) \end{array}$$

Our Patch Classification Problem. Suppose we are given a set $\mathbb{T} \subseteq Pgm \times Pgm$ of patches that are partitioned into correct patches $\mathbb{T}_c$ and incorrect patches $\mathbb{T}_i$. We aim to find an optimal *patch classification formula* ψ^* such that

$$(P_b, P_p) \in \mathbb{T}_c \iff [\![\psi^*]\!](P_b, P_p) = \top$$

or equivalently, $(P_b, P_p) \in \mathbb{T}_i \iff [\![\psi^*]\!](P_b, P_p) = \bot$.

3.2 Extracting Semantic Features

This section illustrates how Prism extracts semantic features from patches.

Semantic Features. First, we define semantic features. A semantic feature $Feature_s : Pgm \times Pgm \to \{\top, \bot\}$ is a predicate that determines whether a patch causes a change in a *semantic property*. To construct these features, we initially considered a wide range of semantic properties that could reflect diverse patch behaviors. For clarity, we organize them into three categories: (1) basic program state properties (variables, values, and memory updates), (2) object-oriented properties (type casting, inheritance relations, and method overriding), and (3) control-flow and exception-related properties (branch conditions, null checks, and various exception types). However, through iterative implementation and empirical evaluation, we found that some of the initial features were either too sparsely observed or too difficult to analyze with sufficient precision and scalability. Moreover, including a large number of uninformative features degraded the performance of our learning algorithm (Section 3.3). Therefore we pruned the feature set using two criteria:

Table 2. Semantic properties within a single program

#	Description
1	Declared methods in the program
2	Defined variables in the program
3	Used variables in the program
4	Unused variables in the program
5	Thrown exceptions
6	Return-related memory locations of each method
7	Memory locations with constants at method exit
8	Methods that always return the same value
9	Occurrence of Null Pointer Exception (NPE)
10	Occurrence of Index Out of Bounds Exception (IOB)
11	Method calls that terminated abnormally

(1) The feature must have high expressiveness in capturing meaningful semantic differences.
(2) The feature must be extractable with reasonable precision and scalability in practice.

The final 11 features listed in Table 2. They cover a wide range of semantic dimensions: variable and function usage (1–4), control and error flow (5, 9–11), data flow (6), and value-related semantics (7, 8). These categories align with the common criteria for program semantics.

We define the changes in these semantic properties between two programs as our semantic features. Specifically, for each semantic property, we consider three types of changes: (1) Gen_i represents the generation of the i-th property by the patch, (2) Del_i denotes its removal, and (3) Remain_i reflects its preservation. Consequently, we obtain a total of 33 semantic features, encompassing three types of changes for each of the 11 property types. In the actual implementation, we refined the 11 semantic properties to yield 66 distinct semantic features. For example, variable-related properties were further categorized by type (e.g., local variable, parameter, class field).

To extract semantic features from a pair of programs (P_b, P_p), Prism first performs static analysis on each program and computes a pair of abstract semantics: $(\widehat{[\![P_b]\!]}, \widehat{[\![P_p]\!]})$. Then, using a checker C_i, it extracts the set of alarms for the i-th property from the analysis results. Finally, it compares the resulting alarm sets to describe changes. Formally, the semantic changes of the i-th semantic property are defined as follows:

$$\begin{aligned}
\mathsf{Gen}_i(P_b, P_p) &\iff \mathsf{C}_i(\widehat{[\![P_p]\!]}) \setminus \mathsf{C}_i(\widehat{[\![P_b]\!]}) \neq \emptyset \\
\mathsf{Del}_i(P_b, P_p) &\iff \mathsf{C}_i(\widehat{[\![P_b]\!]}) \setminus \mathsf{C}_i(\widehat{[\![P_p]\!]}) \neq \emptyset \\
\mathsf{Remain}_i(P_b, P_p) &\iff \mathsf{C}_i(\widehat{[\![P_b]\!]}) \cap \mathsf{C}_i(\widehat{[\![P_p]\!]}) \neq \emptyset
\end{aligned}$$

The definitions of our static analyzer $\widehat{[\![P]\!]}$ and the checker C_i will be discussed later in this section.

Design Principles of Prism's Static Analysis. Because Prism derives semantic features from static analysis, its overall effectiveness critically depends on the underlying analyzer. To support accurate and practical patch classification, the analysis must strive for maximal soundness, accuracy, and scalability:

- **Accuracy:** Insufficient accuracy leads to spurious features, making it harder to distinguish correct from incorrect patches.

- **Soundness:** Unsound analysis may fail to capture actual behavioral changes introduced by patches, leading to missed features.
- **Scalability:** The analysis must efficiently handle a large number of patches generated by APR tools, as well as the complexity of real-world programs.

Balancing these tradeoffs is a fundamental challenge in static analysis. While our goal is to maximize soundness, accuracy, and scalability, full formal guarantees are often impractical in real-world settings. Specifically, our analysis sacrifices completeness and full soundness to improve practical accuracy and scalability. For instance, it applies bounded loop unrolling, which may miss some behaviors but effectively captures most patch-relevant effects. Instead of pursuing full correctness, our analysis is designed to extract a rich set of semantic features that capture patch-induced behavioral changes with high reliability and efficiency. To this end, our analysis is designed to meet the following practical goals:

(1) **Patch-aware disjunctive analysis**: Preserve patch-affected abstract states while aggressively merging irrelevant states to reduce analysis cost.
(2) **Bounded soundeness**: Over-approximate behaviors within a fixed bound by loop unrolling to capture key effects efficiently.
(3) **Modular interprocedural analysis**: Perform a bottom-up, summary-based analysis to enable efficient reuse and incremental re-analysis.

Despite these efforts, the static analysis in Prism is inherently unsound and imprecise by design, which may lead to misclassifications of correct patches (see Section 4.4.3). We now describe the abstract domain used in our analysis.

Abstract Domain. To detect the semantic properties in Table 2, we have designed the following abstract domain:

$$
\begin{array}{lcccl}
d & \in & \mathbb{D}_P & = & \mathbb{L}_P \to \mathcal{P}(State) \\
s & \in & State & = & PC \times Memory \times D_{data} \\
\pi & \in & PC & = & \mathcal{P}(\widehat{Val} \times \prec \times \widehat{Val}) \\
\sigma & \in & Memory & = & Loc \to \widehat{Val} \\
\delta & \in & D_{data} & = & \mathcal{P}(Loc \times Loc) \\
l & \in & Loc & = & Var + (AllocSite \times \widehat{Val}) + (AllocSite \times Var) \\
\ell & \in & AllocSite & \subseteq & \mathbb{L}_P \\
v & \in & \widehat{Val} & = & \{\texttt{null}\} + \mathbb{Z} + AllocSite + Class + \{\bot, \top\}
\end{array}
$$

This domain is based on disjunctive analysis: it computes a table $d \in \mathbb{D}_P$ that maps each program point to a set of reachable abstract states. An abstract state $s \in State$ consists of three components: (1) a set of path conditions $\pi \in PC$, representing binary relations between abstract values; (2) a memory state $\sigma \in Memory$, which maps locations to abstract values; and (3) a set of data dependency relations $\delta \in D_{data}$ between locations. We use δ^* to denote the reflexive transitive closure of δ. A location $l \in Loc$ is either a variable ($x \in Var$), an array element at a specific allocation site and index ($AllocSite \times \widehat{Val}$), or a field access on an object allocated at a particular site ($AllocSite \times Var$). An abstract value $v \in \widehat{Val}$ can be a literal (`null` or $\mathbb{Z}$), an allocation site ($AllocSite$) representing a base address or symbolic parameter for method invocation modeling, an exception class ($Class$) for exception handling, $\bot$ if undefined, and $\top$ if it may represent multiple values. For simplicity, we abstract objects by their allocation sites only, though the implementation also tracks their types.

The abstract transfer function $F_P : \mathbb{D}_P \to \mathbb{D}_P$ is defined as an abstraction of all possible transitions of a program P:

$$F_P(d) = d \sqcup \Big(\bigsqcup_{\ell \in \mathbb{L}_P} \bigsqcup_{\ell' \in \mathsf{next}_P(\ell)} [\ell' \mapsto \bigcup_{s \in d(\ell)} \widehat{[\![\mathsf{stmt}_P(\ell)]\!]}^{\ell}_P(d, s)]$$

To define the abstract semantics, we first describe how to evaluate an l-value. The locations l and l' of l-values lv and lv', respectively, under memory σ are computed by $\mathcal{L}[\![lv]\!] : \mathit{Memory} \to \mathit{Loc}$:

$$\mathcal{L}[\![x]\!](\sigma) = x \qquad \mathcal{L}[\![x[y]]\!](\sigma) = (\sigma(x), \sigma(y)) \qquad \mathcal{L}[\![x.y]\!](\sigma) = (\sigma(x), y)$$

The abstract semantics $\widehat{[\![S]\!]}^{\ell}_P : \mathbb{D}_P \times \mathit{State} \to \mathcal{P}(\mathit{State})$ of a statement S at label ℓ of program P is defined as follows:

$$\begin{array}{rcl}
\widehat{[\![lv := lv']\!]}^{\ell}_P(d, (\pi, \sigma, \delta)) & = & \{(\pi, \sigma[l \mapsto \sigma(l')], \delta \cup \{(l, l')\})\} \\
\widehat{[\![lv := c]\!]}^{\ell}_P(d, (\pi, \sigma, \delta)) & = & \{(\pi, \sigma[l \mapsto c], \delta)\} \qquad \text{where } c \in \mathbb{Z} \cup \{\mathtt{null}\} \\
\widehat{[\![lv := m(x)]\!]}^{\ell}_P(d, (\pi, \sigma, \delta)) & = & \bigcup_i \{(\pi \cup \pi^i_m, \sigma[l \mapsto \sigma^i_m(ret_m)], \delta \cup \delta^i_m \cup \{(l, ret_m)\})\} \\
 & & \qquad \text{where } (\pi^i_m, \sigma^i_m, \delta^i_m) \in d(\ell^e_m)[\ell^s_m \mapsto \sigma(x)] \\
\widehat{[\![lv := \mathtt{new}\ C()]\!]}^{\ell}_P(d, (\pi, \sigma, \delta)) & = & \{(\pi, \sigma[l \mapsto \ell], \delta)\} \\
\widehat{[\![\mathtt{new}\ x[y]]\!]}^{\ell}_P(d, (\pi, \sigma, \delta)) & = & \{(\pi \cup \{\sigma(y), \geq, 0\}, \sigma[x \mapsto \ell, x_{size} \mapsto \sigma(y)], \delta \cup \{(x, y)\})\} \\
\widehat{[\![\mathtt{throw}\ \mathtt{new}\ C]\!]}^{\ell}_P(d, (\pi, \sigma, \delta)) & = & \{(\pi, \sigma[exn \mapsto C], \delta)\} \\
\widehat{[\![\mathtt{assume}(x \prec y)]\!]}^{\ell}_P(d, (\pi, \sigma, \delta)) & = & \begin{cases} \{(\pi \cup \{(\sigma(x), \prec, \sigma(y))\}, \sigma, \delta)\} & \text{if } \mathsf{SAT}(\pi \wedge (\sigma(x) \prec \sigma(y))) \\ \emptyset & \text{otherwise} \end{cases} \\
\widehat{[\![\mathtt{catch}\ C]\!]}^{\ell}_P(d, (\pi, \sigma, \delta)) & = & \begin{cases} \{(\pi, \sigma[exn \mapsto \bot], \delta)\} & \text{if } \sigma(exn) = C \\ \emptyset & \text{otherwise} \end{cases}
\end{array}$$

We assume that all non-catch statements are skipped when $\sigma(exn) \neq \bot$. For a method invocation $m(x)$, we model the parameter of m using its entry label ℓ^s_m as a symbolic value. During bottom-up analysis, we compute the exit states $d(\ell^e_m)$ of m with respect to this symbolic parameter. At the call site, these callee states are instantiated by substituting ℓ^s_m with the actual argument $\sigma(x)$, resulting in states $(\pi^i_m, \sigma^i_m, \delta^i_m) \in d(\ell^e_m)[\ell^s_m \mapsto \sigma(x)]$. For `assume` statements, we accumulate path conditions. If the path becomes unsatisfiable (i.e., $\neg\mathsf{SAT}(\pi \wedge (\sigma(x) \prec \sigma(y)))$), the state is pruned. For `catch` statements, we reset exn to $\bot$ when it contains an exception of type C, indicating that it is handled; otherwise, the state is pruned. The abstract semantics of a program P is defined as the least fixed point of the transfer function F_P:

$$\widehat{[\![P]\!]} = \lim_{i \to \infty} (F_P)^i(d^s)$$

where $d^s = \bigsqcup_{m \in \mathit{Mthd}_P} [\ell^s_m \mapsto \{(\emptyset, [p_m \mapsto \ell^s_m], \emptyset)\}]$ is the initial abstract state, where each method's entry label ℓ^s_m is assigned to its formal parameter p_m.

Selective State Merging for Patch-Aware Analysis. Note that the join operator $\sqcup$ performs a weak update, accumulating all reachable states at each label without merging. While this ensures precision, it leads to a rapid growth in the number of states, causing scalability issues. To mitigate this, we employ a state merging strategy that balances precision with efficiency. To preserve soundness, the merge of two abstract states $s_1 = (\pi_1, \sigma_1, \delta_1)$ and $s_2 = (\pi_2, \sigma_2, \delta_2)$ is defined as:

- Path conditions and data dependencies are merged via set union: $\pi = \pi_1 \cup \pi_2$, $\delta = \delta_1 \cup \delta_2$.
- Memory is merged pointwise: $(\sigma_1 \sqcup \sigma_2)(l) = \sigma_1(l) \sqcup \sigma_2(l)$.

```
1   int x = 1;
2 - if (C) x = foo();
3 + if (C) x = 0;
4   if (x < 0)
5     goo(y)
```

(a) a patch program p

$\mathbb{L}_p$	$d(\ell)$	$\sigma(x)$
ℓ_1	$\{\sigma\}$	1
ℓ_3	$\{\sigma\}$	0
ℓ_4	$\{\sigma\}$	$\top$
ℓ_5 : reachable ($\because \sigma(x) = \top$)		

(b) analysis results (merged)

$\mathbb{L}_p$	$d(\ell)$	$\sigma(x)$	$\sigma_p(x)$
ℓ_1	$\{\sigma\}$	1	$\bot$
ℓ_3	$\{\sigma_p\}$	$\bot$	0
ℓ_4	$\{\sigma, \sigma_p\}$	1	0
ℓ_5 : unreachable ($\because \sigma(x), \sigma_p(x) \geq 0$)			

(c) analysis results (unmerged)

Fig. 3. An example patch and analysis results illustrating the effect of patch-aware state merging. We only show the memory (σ) of abstract states $d(\ell) = (\pi, \sigma, \delta)$ for brevity.

Merging two different concrete values yields $\top$ (e.g., $1 \sqcup 0 = \top$), which represents uncertainty. While sound, such merging may obscure meaningful semantic differences.

Figure 3 illustrates the importance of selective merging. The patch replaces the original `if` branch at line 2. In the merged result (b), all states after line 3 are combined, yielding $\sigma(x) = \top$. This causes the analysis to conservatively assume that the condition `x < 0` at line 4 may hold, making line 5 reachable. However, this is imprecise: in the patched program, x is always set to 1 or 0, making the condition unsatisfiable. To avoid this imprecision, Prism preserves states that are *affected by the patch*, as shown in (c). It prevents merging of such patch-relevant states σ_p arising from the inserted line. As a result, the analysis retains the precision needed to correctly determine that line 5 is unreachable in all preserved paths.

Formally, we consider two states $s_1 = (\pi_1, \sigma_1, \delta_1)$ and $s_2 = (\pi_2, \sigma_2, \delta_2)$ to be mergeable as follows:

$$\text{mergable}(s_1, s_2) = \begin{cases} \text{SAT}(\pi_1 \wedge \pi_2) & \text{if } T(s_1) \cup T(s_2) \subseteq \mathbb{L}_{P_b} \cap \mathbb{L}_{P_p} \\ T(s_1) \cap \mathbb{L}_{Patch} = \emptyset \wedge T(s_2) \cap \mathbb{L}_{Patch} = \emptyset & \text{otherwise} \end{cases}$$

where $T : State \to \mathbb{L}_P$ denotes labels traversed in the execution trace leading to state s. Intuitively, if both states have not encountered any patch-related program points yet, we merge them only if their path conditions are satisfiable (i.e., conventional SAT-based merging). Otherwise, we conservatively merge only states that are unaffected by the patch. Given this, states at ℓ is updated as:

$$d[\ell \mapsto \bigcup_{\substack{s_1, s_2 \in d(\ell) \\ \text{mergable}(s_1, s_2)}} (s_1 \sqcup s_2) \cup \bigcup_{\substack{s \in d(\ell) \\ \neg \exists s' \in d(\ell). \text{mergable}(s, s')}} \{s\}]$$

Alarm Extraction. We represent the i-th properties in Table 2 as alarms, which include a variable (*Var*), a location (*Loc*), a method name (*Mthd*), a class name (*Class*), or a program point ($\mathbb{L}$):

$$a_i \in Alarm = Var + Loc + Mthd + Class + \mathbb{L}$$

Each alarm $a_i \in Alarm$ has a corresponding checker $\text{C}_i : \mathbb{D}_P \to \mathcal{P}(Alarm)$ that extracts alarms from abstract semantics. Table 3 lists the 11 alarm types and the definitions of each checker, where $d(\ell) = \emptyset$ means ℓ is unreachable. Since our analysis allows $\top$, alarms triggered under $\top$-related conditions may be spurious (i.e., semantic features that do not actually occur). To mitigate this issue, each checker in Table 3 is designed conservatively to capture only *highly probable* alarms. For example, an NPE is reported only if a state definitively assigns `null` to an array. While this conservative approach avoids false positives, it may also miss valid alarms. The following section describes how we address this limitation.

Table 3. List of alarm types and their corresponding checkers $\mathsf{C}_i(d)$ where $d = \widehat{[\![P]\!]}$

No	Alarm type	Definition of $\mathsf{C}_i(d)$
1	Declared methods	$Mthd_P$
2	Defined variables	$\{x \in Var_P \mid \exists \ell_d \in \mathbb{L}_P.d(\ell_d) \neq \emptyset \wedge x \in \mathsf{def}(\ell_d)\}$
3	Used variables	$\{x \in Var_P \mid \exists \ell_u \in \mathbb{L}_P.d(\ell_u) \neq \emptyset \wedge x \in \mathsf{use}(\ell_u)\}$
4	Unused variables	$\{x \in Var_P \mid \exists \ell_d \in \mathbb{L}_P.\forall \ell_u \in \mathbb{L}_P.d(\ell_u) = \emptyset \vee x \in \mathsf{def}(\ell_d) \setminus \mathsf{use}(\ell_u)\}$
5	Thrown exceptions	$\{C \mid \exists m \in Mthd_P.(\pi, \sigma, \delta) \in d(\ell^e_m) \wedge \sigma(exn_m) \neq \bot\}$
6	Return-related locs	$\{l \in Loc \mid \exists m \in Mthd_P.\exists(\pi, \sigma, \delta) \in d(\ell^e_m).(ret_m, l) \in \delta^*\}$
7	Constant locs	$\{l \in Loc \mid \exists m \in Mthd_P.\exists c \in \mathbb{Z} \cup \{\mathtt{null}\}.\forall(\pi, \sigma, \delta) \in d(\ell^e_m).\sigma(l) = c \vee (\sigma(l), =, c) \in \pi\}$
8	Constant methods	$\{m \in Mthd_P \mid \exists c \in \mathbb{Z} \cup \{\mathtt{null}\}.\forall(\pi, \sigma, \delta) \in d(\ell^e_m).\sigma(ret_m) = c \vee (\sigma(ret_m), =, c) \in \pi\}$
9	NPE lines	$\{\ell \in \mathbb{L}_P \mid \mathsf{E}(\ell) \in \{x[y], x.y\} \wedge \exists(\pi, \sigma, \delta) \in d(\ell).\sigma(x) = \mathtt{null} \vee (\sigma(x), =, \mathtt{null}) \in \pi\}$
10	IOB lines	$\{\ell \in \mathbb{L}_P \mid \mathsf{E}(\ell) = x[y] \wedge \exists(\pi, \sigma, \delta) \in d(\ell).\sigma(y) > \sigma(x_{size}) \vee (\sigma(y), >, \sigma(x_{size})) \in \pi\}$
11	Exception call lines	$\{\ell \in \mathbb{L}_P \mid \mathsf{E}(\ell) = m(x) \wedge \exists(\pi, \sigma, \delta) \in d(\ell).\sigma(exn_m) \neq \bot\}$

Leveraging Test Execution Information. Prism leverages test execution information to detect true alarms observed during testing. Assuming the following code (left) which gets the first element of the given integer array a and two test cases (right):

```
1 int foo (int[] a) {
2   return a[0];
3 }
```

Test No.	Input	Expected	Output
1	[1]	1	1
2	null	-1	NPE

The developer assumes that if the input is null, the method should return -1, as expressed in the second test. However, the method at line 2 runs without null-handling for a and directly dereferences the array, causing an NPE during the execution of the second test case. However, our static analyzer cannot catch this NPE because it considers a to be either null or non-null at that line. Therefore, it misses a true alarm that is easily captured during testing.

To detect such true alarms, we leverage test execution information, specifically the error occurrence information that can be easily obtained from stack traces. The test execution information Test_P of a program P is defined as follows:

$$\mathsf{Test}_P : \mathbb{L}_P \to \{\mathrm{NPE}, \mathrm{IOB}, \mathrm{ExnCall}, \cdot\}$$

It records the type of error (NPE, IOB, or ExnCall), or indicates no error ($\cdot$) for each line. Using the test execution information Test_P, we define the improved version of the checker $\mathsf{C}_i^{\mathsf{Test}_P}$:

$$\mathsf{C}_i^{\mathsf{Test}_P}(d) = \mathsf{C}_i(d) \cup A_i^{\mathsf{Test}_P}$$

where $A_i^{\mathsf{Test}_P}$ is the set of additional alarms obtained from test execution information. Since Test_P contains only error occurrence information, $A_i^{\mathsf{Test}_P}$ is defined only for the error-related alarms ($i = 9, 10, 11$) and is empty for the rest ($i = 1$ to 8):

$$\begin{aligned} A_9^{\mathsf{Test}_P} &= \{\ell \in \mathbb{L}_P \mid \mathsf{Test}_P(\ell) = \mathrm{NPE} \wedge \mathsf{E}(\ell) \in \{x[y], x.y\}\} \\ A_{10}^{\mathsf{Test}_P} &= \{\ell \in \mathbb{L}_P \mid \mathsf{Test}_P(\ell) = \mathrm{IOB} \wedge \mathsf{E}(\ell) = x[y]\} \\ A_{11}^{\mathsf{Test}_P} &= \{\ell \in \mathbb{L}_P \mid \mathsf{Test}_P(\ell) = \mathrm{ExnCall} \wedge \mathsf{E}(\ell) = x.m()\} \end{aligned}$$

From now on, we omit the superscript Test_P; i.e., we use the improved checker $\mathsf{C}_i^{\mathsf{Test}_P}$ as the default.

Leveraging Analysis Result of Original Program. To analyze the patched program more accurately, Prism leverages original analysis results. Consider two different patches for an NPE bug in Figure 4. The patch in Figure 4a correctly fixes the NPE by modifying the null-handling condition. In contrast, the patch in Figure 4b partially fixes the NPE, addressing it only when $x \neq 0$. Prism considers both patches as NPE-removal patches because they eliminate the fact that a is

```
1 int foo (int[] a, int x) {
2 - if (a == null)
3 + if (a != null)
4     return a[x];
5   else return 0;
6 }
```

$\mathbb{L}$	$\sigma(a)$	π
ℓ_4	α_a	$\{(\alpha_a, =, \texttt{null})\}$

Analysis result before patch

$\mathbb{L}$	$\sigma(a)$	π
ℓ_4	α_a	$\{(\alpha_a, \neq, \texttt{null})\}$

Analysis result after patch

(a) A patch properly addressing the NPE

```
1 int foo (int[] a, int x) {
2 - if (a == null)
3 + if (x == 0)
4     return a[x];
5   else return 0;
6 }
```

$\mathbb{L}$	$\sigma(a)$	π
ℓ_4	α_a	$\{(\alpha_a, =, \texttt{null})\}$

Analysis result before patch

$\mathbb{L}$	$\sigma(a)$	π
ℓ_4	α_a	$\{(\alpha_x, =, 0)\}$

Analysis result after patch

(b) A patch that does not completely resolve the NPE

Fig. 4. Two patches for the same NPE bug (α_a and α_x represent the formal parameters a and x respectively.)

definitely `null` at line 4. However, we can infer that the second patch still has an NPE based on the following evidences: (1) an alarm indicating a potential NPE before the patch, and (2) the object `a` related to this alarm *remains unchanged* after the patch.

To reflect this inference process, we define the improved checker C_i^p for the patched program:

$$C_i^p(d_p) = C_i(d_p) \cup A_i^{P_p}$$

The additional alarms $A_i^{P_p}$ of the patched program P_p are based on the fact that the values related to the original alarms remain unchanged. Therefore, $A_i^{P_p}$ is defined only for the value-related alarms ($i = 7$ to 11) and is empty for the rest ($i = 1$ to 6):

$$A_7^{P_p} = \{x \in C_7(d_b) \mid \exists m \in Mthd_P.\mathcal{U}_{P_p}(\ell_m^e, x)\}$$
$$A_8^{P_p} = \{m \in C_8(d_b) \mid \mathcal{U}_{P_p}(\ell_m^e, ret_m)\}$$
$$A_9^{P_p} = \{\ell \in C_9(d_b) \mid \mathsf{E}(\ell) \in \{x[y], x.y\} \wedge \mathcal{U}_{P_p}(\ell, x) \wedge \exists(\pi, \sigma, \delta) \in d_p.(\sigma(x), \neq, \texttt{null}) \notin \pi\}$$
$$A_{10}^{P_p} = \{\ell \in C_{10}(d_b) \mid \mathsf{E}(\ell) = x[y] \wedge \mathcal{U}_{P_p}(\ell, x) \wedge \mathcal{U}_{P_p}(\ell, y) \wedge \exists(\pi, \sigma, \delta) \in d_p.(\sigma(x_L), \geq, \sigma(y)) \notin \pi\}$$
$$A_{11}^{P_p} = \{\ell \in C_{11}(d_b) \mid \mathsf{E}(\ell) = m(x) \wedge \mathcal{U}_{P_p}(\ell, x)\}$$

Each set $A_i^{P_p}$ contains alarms from the original analysis $C_i(d_b)$ that remain unaddressed and whose related variables are unchanged after the patch. The predicate $\mathcal{U}_P : \mathbb{L}_P \times Var \to \{\top, \bot\}$ checks whether a variable x is unpatched at the label ℓ of the program P:

$$\mathcal{U}_P(\ell, x) \equiv \forall(\pi, \sigma, \delta) \in d(\ell).\forall \ell_p \in \mathbb{L}_{Patch}.\neg\exists y \in \mathsf{def}(\ell_p).(x, y) \in \delta^*$$

where $d = \widehat{[\![P]\!]}$. A variable x is considered unpatched at ℓ when no relations between directly patched variables and x.

3.3 Learning Algorithm

In this section, we introduce our learning algorithm for finding a patch classifier. Given a set of atomic features $\mathbb{A} \subseteq Feature$, a training set $\mathbb{T} \subseteq Pgm \times Pgm$, and a user-provided threshold $\gamma \in [0, 1]$,

the algorithm produces a boolean formula ψ in disjunctive normal form (DNF).

$$\psi = \bigvee_{i=1}^{m} (\bigwedge_{j=1}^{n_i} l_{i,j})$$

where each literal $l_{i,j}$ is either a boolean constant ($\top$ or $\bot$), atomic feature $f_i \in \mathbb{A}$, or its negation $\neg f_i$. We represent a DNF formula ψ as a set of conjunctive clauses:

$$\psi = \{c_1, c_2, \ldots, c_m\}$$

where each conjunctive clause c_i is a set of literals:

$$c_i = \{l_{i,1}, l_{i,2}, \ldots, l_{i,n_i}\}$$

Learning Objective. Since finding an optimal solution ψ^* is generally infeasible in practice, our learning algorithm instead aims to find a sub-optimal solution $\tilde{\psi}$. As we emphasized earlier, for APCC techniques to be practical in real-world APR environments, they should preserve as many correct patches as possible while identifying incorrect ones. To achieve this goal, our algorithm searches for a formula $\tilde{\psi}$ that maximizes the *incorrect patch detection ratio* (IDR) while ensuring that the *correct patch preserving ratio* (CPR) is at least the user-provided threshold γ:

$$\tilde{\psi} = \operatorname*{argmax}_{\psi} IDR_{\mathbb{T}}(\psi) \quad \text{s.t.} \quad CPR_{\mathbb{T}}(\psi) \geq \gamma$$

where the IDR and CPR of ψ on the training set $\mathbb{T}$ are defined respectively as follows:

$$IDR_{\mathbb{T}}(\psi) = \frac{\left|\left\{(P_b, P_p) \in \mathbb{T}_i \mid [\![\psi]\!](P_b, P_p) = \bot\right\}\right|}{|\mathbb{T}_i|} \quad CPR_{\mathbb{T}}(\psi) = \frac{\left|\left\{(P_b, P_p) \in \mathbb{T}_c \mid [\![\psi]\!](P_b, P_p) = \top\right\}\right|}{|\mathbb{T}_c|}$$

Overall Algorithm. Algorithm 1 outlines the overall process. It takes the following six inputs:

(1) A set of atomic features $\mathbb{A}$.
(2) A training set $\mathbb{T}$ consisting of patches.
(3) A CPR threshold γ, which the resulting formula must satisfy on $\mathbb{T}$.
(4) A hyperparameter K_{init} denoting the initial candidate size.
(5) A decay rate $\delta \in (0, 1)$ for adjusting the candidate size.
(6) A generalization threshold k for enlarging formulas after repeated failures.

The algorithm performs a state search using boolean formulas as workset elements. Initially (lines 1-2), the workset W is empty, and the solution candidate $\tilde{\psi}$ is set to an empty set with zero IDR. The non-improving iteration counter cnt_{ni} is initialized to zero, tracking the number of consecutive iterations without improvement. The candidate selection threshold K is set to K_{init}.

Lines 3-23 constitute the main loop of the algorithm. At the beginning of each iteration, selects the top K candidates with the highest IDR values from the current workset (line 8). These selected formulas are then expanded using the $\textsc{Specify}_{\mathbb{A}}$ function to generate new candidate formulas:

$$\textsc{Specify}_{\mathbb{A}}(\psi) = \{\psi[c_i \mapsto c_i \cup \{f_i\}] \mid c_i \in \psi \wedge f_i \in \mathbb{A} \cup \neg\mathbb{A}\}$$

where $\neg\mathbb{A} = \{\neg f_i \mid f_i \in \mathbb{A}\}$ represents the negation of all atomic features. The $\textsc{Specify}_{\mathbb{A}}$ function constructs new formulas by extending each conjunctive clause in the original formula with an additional literal. Note that the newly generated formulas $\psi' \in \textsc{Specify}_{\mathbb{A}}(\psi)$ represent a subset of the correct patches captured by ψ. Consequently, the new formula ψ' exhibits a monotonic increase in IDR and a monotonic decrease in CPR. This property ensures that the algorithm continuously refines its search, selecting higher-IDR candidates while pruning those with CPR below γ (line 9). After refining the workset, the algorithm reduces the candidate selection threshold K (line

Algorithm 1 Algorithm for learning a patch classification formula

Input: Atomic features $\mathbb{A}$, training set $\mathbb{T}$, CPR threshold γ, initial candidate size K_{init}, decay rate δ, generalization threshold k

Output: A learned formula $\tilde{\psi}$ with a CPR greater than γ on the training set $\mathbb{T}$

1: $W \leftarrow \emptyset$
2: $\tilde{\psi}, idr, cnt_{ni}, K \leftarrow \emptyset, 0, 0, K_{init}$
3: **repeat**
4: **if** $W = \emptyset$ **then**
5: $W \leftarrow \{(\tilde{\psi} \cup \{\{f_i\}\}) \mid f_i \in \mathbb{A} \cup \neg\mathbb{A}\}$
6: $IsUpdated \leftarrow False$
7: $K = \min(K, |W|)$
8: $\Psi = \{\psi \in W \mid IDR_{\mathbb{T}}(\psi)$ is among the top K highest values in $W\}$
9: $W = \{\psi' \mid \psi \in \Psi \wedge \psi' \in \textsc{Specify}_{\mathbb{A}}(\psi) \wedge CPR_{\mathbb{T}}(\psi') \geq \gamma\}$
10: $K \leftarrow K * \delta$
11: **for** $\psi' \in W$ **do**
12: $idr' \leftarrow IDR_{\mathbb{T}}(\psi')$
13: **if** $idr' > idr$ **then**
14: $\tilde{\psi}, idr \leftarrow \psi', idr'$
15: $IsUpdated \leftarrow True$
16: $cnt_{ni} \leftarrow 0$
17: **if** $\neg IsUpdated$ **then**
18: $cnt_{ni} \leftarrow cnt_{ni} + 1$
19: **if** $cnt_{ni} > k$ **then**
20: $W \leftarrow \{\psi' \mid \psi \in (W \cup \Psi) \wedge \psi' \in \textsc{Generalize}_{\mathbb{A}}(\psi)\}$
21: $cnt_{ni}, K \leftarrow 0, K_{init}$
22: **until** timeout
23: **return** $\tilde{\psi}$

10). Intuitively, this encourages diverse exploration in the early stages of search and gradually prioritizes high-quality candidates over iterations.

If any new candidate ψ' has a higher IDR than the current solution $\tilde{\psi}$, the solution is updated (lines 11-16). In this case, the update flag *IsUpdated* is set to *True*, and the no-improvement counter cnt_{ni} is reset to zero. If no improvement occurs, the counter cnt_{ni} is incremented (lines 17-18). Once cnt_{ni} exceeds the threshold k, the algorithm generalizes the current workset by expanding the extracted promising candidates:

$$\textsc{Generalize}_{\mathbb{A}}(\psi) = \{\psi \cup \{a\} \mid a \in \mathbb{A} \cup \neg\mathbb{A}\}$$

A new formula $\psi' \in \textsc{Generalize}_{\mathbb{A}}(\psi)$ always represents a superset of the correct patches described by the given formula ψ. Consequently, this results in a monotonic increase in CPR and a monotonic decrease in IDR, promoting exploration by allowing the learning algorithm to discover new formulas. After generalization, the counter cnt_{ni} and the candidate selection threshold K are reinitialized, and the search resumes. This process continues until the timeout is reached (line 22). Finally, the algorithm returns the solution $\tilde{\psi}$ (line 23).

4 Evaluation

In this section, we experimentally evaluate Prism to answer the following research questions.

- **RQ1. Impact on End-to-End APR**: How does Prism improve APR performance while reducing human effort in patch validation?
- **RQ2. Patch Classification Performance**: How effectively does Prism classify patches?
- **RQ3. Ablation Study**: Are the semantic features and learning algorithm essential for the performance of Prism?

Comparison Target Selection Criteria. We compared Prism with three APCC techniques, each representing a distinct category: PATCH-SIM [52] (dynamic), ODS [56] (syntactic), and Shibboleth [10] (hybrid). We did not evaluate other methods based on syntactic features, such as those by Lin et al. [24] and Tian et al. [45, 46], because although they employ different feature sets and models than ODS, they belong to the same syntactic category and their performance does not differ significantly from that of ODS [45, 46]. In our experiments, we also excluded Opad [55], a tool originally designed for C programs that relies heavily on specialized vulnerability detectors. We found that, in a Java environment, no suitable counterparts exist for them[1]. We were unable to evaluate the LLM-based approach [61]; although the benchmark and code for reproducing the results are available, applying it to a new dataset is challenging. We did not evaluate approaches that are not fully automated; for instance, those that require user interaction [13] or developer-provided fixes as ground truth [22, 51]. In addition, we implemented a rule-based classifier (Rule-Based) that reflects intuitive characteristics of correct patches. These rules were manually designed based on patterns observed in our learned formula (see Section 4.4.2). We obtained all the tools from the links provided in their respective papers. PATCH-SIM normally leverages both developer-provided tests and tests generated by Randoop. When Randoop fails to generate tests, we ran PATCH-SIM using only the developer tests. When any technique failed to run due to internal errors, we assumed that the tool classified the patch as correct. This assumption is based on the fact that APCC techniques are primarily employed to filter out patches marked as incorrect.

Datasets for Patch Classification. To ensure robust training and evaluation, we constructed a new dataset by integrating and refining data from prior studies [27, 47, 52, 58]. The dataset consists of patches generated by various APR tools for Defects4J, a widely used benchmark containing real bugs from six open-source Java projects: Chart, Closure, Lang, Math, Time, and Mockito. Because the original datasets predominantly contained incorrect patches, we augmented 388 correct patches (excluding 7 deprecated ones) to balance the distribution. The initial dataset contained 4,019 patches. We then removed non-compilable and implausible patches, which are generally not considered valid inputs for APCC techniques. Additionally, we eliminated syntactically redundant patches to improve overall dataset quality. To ensure labeling consistency, we manually re-labeled all patches by checking their semantic equivalence to the developer's patch (ground truth), following the standard criterion used in APR and APCC research. When this was difficult to apply due to ambiguity, we referred to the labeling guidelines proposed by Liu et al. [27]. If neither approach was sufficient, we used the labels provided in the original APR papers. The finalized dataset comprises 522 correct patches and 1,307 incorrect patches.

Patch Classifier Training Setup. To eliminate data leakage and properly evaluate generalization, we used a project-by-project approach on Defects4J to train a patch classifier. Specifically, we constructed six distinct training datasets, each excluding patches from one project. For instance, when testing on the Chart project, the training set consisted solely of patches from the remaining projects. This approach guarantees that there is no overlap between the training and testing sets. As we assembled a new dataset, we re-trained the patch classifiers for both ODS and Shibboleth.

[1]Although previous studies have evaluated Opad using *Randoop*, we observed that this setting drastically reduced its effectiveness, yielding an incorrect patch filtering rate below 5%.

Table 4. Benchmark Table: Performance metrics for the ten APRs. # Bugs with Top-k Fixes: The number of bugs for which a correct patch was successfully returned as the first plausible patch (Top-1), within the first five plausible patches (Top-5), or at least once among all generated patches (Top-∞). # Generated Patches: The total number of correct and incorrect patches generated by each APR. # Patches to Review: The number of patches a developer needs to review before encountering the first correct patch (To 1st Fix) or all correct patches (To All Fixes).

APR	# Bugs with Top-k Fixes			# Generated Patches		# Patches to Review	
	Top-1	Top-5	Top-∞	Correct	Incorrect	To 1st Fix	To All Fixes
TBar [25]	29	33	37	51	278	142	219
AlphaRepair [49]	14	18	22	67	342	250	348
CoCoNuT [31]	9	10	10	38	70	11	72
CURE [16]	13	18	18	27	132	28	49
Edits [5]	2	2	2	2	8	2	2
Recoder [62]	28	29	30	33	63	37	44
RewardRepair [59]	25	28	29	107	170	40	179
SelfAPR [57]	29	32	33	224	175	54	291
SequenceR [4]	13	16	17	25	88	29	58
SimFix [15]	14	15	15	15	22	16	16
Total	176	201	213	589	1348	609	1278

For ODS, we utilized the provided XGBoost [3] model with the hyperparameters from the original work [56]. For Shibboleth, we employed a random forest classifier [11] as described in the original work; however, since the exact hyperparameter settings were not provided, our re-trained model may not fully replicate the reported performance.

Implementation of Prism. We implemented the static analyzer and semantic feature extractor of Prism in 9,356 lines of OCaml code built on top of Infer [12]. Our learning algorithm and patch classification module were developed in 363 lines of Python. The CPR threshold γ for the learning algorithm was experimentally set to 90%, which maximizes the filtering of incorrect patches without reducing the APR fix rate. The learning hyperparameters K_{init}, δ, and k in Algorithm 1 were set to 1000, 0.75, and 2, respectively. We set a time budget of 20 minutes for the static analysis and a 3-minute limit for the learning algorithm. All experiments were conducted on an Ubuntu machine equipped with an AMD Ryzen Threadripper 3990X 64-Core Processor and 256GB of memory.

4.1 RQ1: Impact on End-to-End APR

We begin by revisiting the motivation of this work: developing an APCC technique that improves APR performance while reducing human validation effort.

Evaluation Settings. Table 4 details the performance metrics for the ten APR techniques evaluated in this experiment. To evaluate APCC techniques, we first ran two widely evaluated APR tools: (1) TBar [25] (a template-based approach) and (2) AlphaRepair [49] (an LLM-based approach) on all 388 bugs from Defects4J, with each bug given a 5-hour time budget. To further demonstrate the generality of Prism, we supplemented this data with patches generated by eight additional APR techniques from recent studies [37]. The correctness of all generated patches was manually verified using the criteria described earlier. We were unable to obtain data from the latest LLM-based APR (e.g., ChatRepair [50]) because the data and code are not publicly available, and the practical use of LLM-based patch generation requires substantial costs.

Table 5. Impact of APCC techniques on APR effectiveness and the human effort required for patch validation. Changes due to APCC application are visually represented: 🔴 indicates a positive effect, while 🔵 indicates a negative effect. "# Bugs with Top-k Fixes" (Top-1, Top-5, Top-∞) measures APR effectiveness. "# Patches to Review" (To 1st Fix, To All Fixes) measures the number of patches to examine before finding the first or all correct patches. "# Remaining Patches" (Correct, Incorrect) indicates how many patches remain after applying APCC techniques, indirectly reflecting APR effectiveness (Correct) and review effort (Incorrect).

Settings		# Bugs with Top-k Fixes						# Remaining Patches				# Patches to Review			
APR	APCC	Top-1		Top-5		Top-∞		Correct		Incorrect		To 1st Fix		To All Fixes	
AlphaRepair	PATCH-SIM	9	▼5	12	▼6	15	▼7	24	▼43	255	▼87	236	▼14	247	▼101
	ODS	6	▼8	7	▼11	7	▼15	12	▼55	32	▼310	8	▼242	18	▼330
	Shibboleth	11	▼3	17	▼1	22	-	67	-	342	-	140	▼110	256	▼92
	Rule-Based	14	-	17	▼1	20	▼2	61	▼6	103	▼239	46	▼204	123	▼225
	Prism	15	▲1	18	-	22	-	64	▼3	298	▼44	241	▼9	319	▼29
CoCoNuT	PATCH-SIM	8	▼1	8	▼2	8	▼2	36	▼2	64	▼6	8	▼3	69	▼3
	ODS	3	▼6	4	▼6	4	▼6	11	▼27	10	▼60	5	▼6	12	▼60
	Shibboleth	5	▼4	10	-	10	-	38	-	70	-	16	▲5	74	▲2
	Rule-Based	9	-	10	-	10	-	38	-	37	▼33	11	-	41	▼31
	Prism	9	-	10	-	10	-	37	▼1	36	▼34	11	-	40	▼32
CURE	PATCH-SIM	9	▼4	12	▼6	12	▼6	19	▼8	118	▼14	18	▼10	32	▼17
	ODS	2	▼11	3	▼15	3	▼15	4	▼23	33	▼99	4	▼24	5	▼44
	Shibboleth	13	-	15	▼3	18	-	27	-	132	-	50	▲22	65	▲16
	Rule-Based	15	▲2	17	▼1	17	▼1	26	▼1	91	▼41	19	▼9	36	▼13
	Prism	15	▲2	18	-	18	-	27	-	91	▼41	21	▼7	38	▼11
Edits	PATCH-SIM	2	-	2	-	2	-	2	-	8	-	2	-	2	-
	ODS	0	▼2	0	▼2	0	▼2	0	▼2	0	▼8	0	▼2	0	▼2
	Shibboleth	2	-	2	-	2	-	2	-	8	-	2	-	2	-
	Rule-Based	2	-	2	-	2	-	2	-	8	-	2	-	2	-
	Prism	2	-	2	-	2	-	2	-	8	-	2	-	2	-
Recoder	PATCH-SIM	19	▼9	20	▼9	21	▼9	22	▼11	57	▼6	28	▼9	33	▼11
	ODS	7	▼21	9	▼20	9	▼21	9	▼24	24	▼39	13	▼24	13	▼31
	Shibboleth	22	▼6	28	▼1	30	-	33	-	63	-	48	▲11	58	▲14
	Rule-Based	26	▼2	27	▼2	38	▼2	31	▼2	38	▼25	35	▼2	38	▼6
	Prism	28	-	29	-	30	-	33	-	41	▼22	37	-	40	▼4
RewardRepair	PATCH-SIM	18	▼7	21	▼7	22	▼7	82	▼25	158	▼12	33	▼7	143	▼36
	ODS	9	▼16	11	▼17	11	▼18	32	▼75	53	▼117	15	▼25	37	▼142
	Shibboleth	20	▼5	27	▼1	29	-	107	-	170	-	61	▲21	197	▲18
	Rule-Based	25	-	28	-	28	▼1	101	▼6	115	▼55	33	▼7	136	▼43
	Prism	26	▲1	29	▲1	29	-	106	▼1	117	▼53	34	▼6	141	▼38
SelfAPR	PATCH-SIM	22	▼7	25	▼7	26	▼7	144	▼80	171	▼4	47	▼7	209	▼82
	ODS	12	▼17	13	▼19	13	▼20	70	▼154	56	▼119	14	▼40	87	▼204
	Shibboleth	27	▼2	32	-	33	-	224	-	175	-	62	▲8	299	▲8
	Rule-Based	27	▼2	30	▼2	31	▼2	214	▼10	158	▼17	52	▼2	275	▼16
	Prism	29	-	32	-	33	-	224	-	165	▼10	54	-	285	▼6
SequenceR	PATCH-SIM	6	▼7	9	▼7	10	▼7	15	▼10	79	▼9	22	▼7	47	▼11
	ODS	3	▼10	3	▼13	4	▼13	4	▼21	22	▼66	9	▼20	9	▼49
	Shibboleth	12	▼1	16	-	17	-	25	-	88	-	35	▲6	65	▲7
	Rule-Based	14	▲1	15	▼1	16	▼1	24	▼1	52	▼36	24	▼5	36	▼22
	Prism	15	▲2	16	-	17	-	25	-	55	▼33	25	▼4	39	▼19
SimFix	PATCH-SIM	12	▼2	12	▼3	12	▼3	12	▼3	20	▼2	12	▼4	12	▼4
	ODS	3	▼11	3	▼12	3	▼12	3	▼12	1	▼21	3	▼13	3	▼13
	Shibboleth	15	▲1	15	-	15	-	15	-	22	-	15	▼1	15	▼1
	Rule-Based	13	▼1	14	▼1	14	▼1	14	▼1	13	▼9	15	▼1	15	▼1
	Prism	14	-	15	-	15	-	15	-	17	▼5	16	-	16	-
TBar	PATCH-SIM	22	▼7	26	▼7	28	▼9	36	▼15	195	▼83	84	▼58	153	▼66
	ODS	13	▼16	14	▼19	15	▼22	15	▼36	46	▼232	29	▼113	29	▼190
	Shibboleth	24	▼5	32	▼1	37	-	51	-	278	-	126	▼16	275	▲56
	Rule-Based	29	-	32	▼1	34	▼3	44	▼7	92	▼186	49	▼93	66	▼153
	Prism	32	▲3	34	▲1	37	-	49	▼2	191	▼87	96	▼46	166	▼53
Total	PATCH-SIM	127	▼49	147	▼54	156	▼57	392	▼197	1125	▼223	490	▼119	947	▼331
	ODS	58	▼118	67	▼134	69	▼144	160	▼429	277	▼1071	100	▼509	213	▼1065
	Shibboleth	152	▼24	191	▼10	213	-	589	-	1348	-	555	▼54	1306	▲28
	Rule-Based	174	▼2	192	▼9	200	▼13	555	▼34	707	▼641	286	▼323	768	▼510
	Prism	185	▲9	203	▲2	213	-	582	▼7	1019	▼329	537	▼72	1086	▼192

For each bug, we assumed that the baseline ranking is determined by the order in which an APR tool generates patches, and then applied each APCC technique to this ranked sequence. When an APCC technique functions as a patch correctness classifier (as in PATCH-SIM, ODS, Rule-Based, and Prism), it removes patches labeled as incorrect from the ranking. In contrast, for Shibboleth, which re-ranks the given patch sequence, the resulting ranking is used directly. We measured the impact of APCC by comparing the above metrics before and after its application.

To evaluate the impact of APCC techniques on APR, we considered two aspects:

(1) **Repair Capability:** How APCC influences the number of bugs for which a correct patch is returned as the first plausible patch (Top-1), within the first five plausible patches (Top-5), or at least once among all generated patches (Top-∞).
(2) **Human Effort for Patch Validation:** How APCC affects the total number of incorrect patches and the number of patches a developer must review before encountering the first correct patch (To 1st Fix) or all correct patches (To All Fixes).

Results. Table 5 clearly shows that only Prism achieves improvements in both APR repair capability and the reduction of human effort for patch validation. In contrast, while other techniques reduce the number of patches that need to be reviewed, they also substantially decrease the number of correct patches detected. In essence, these methods fail to meet the practical requirement of preserving correct patches in real-world APR, where reducing review effort should not come at the expense of repair effectiveness. Notably, when applied to TBar, Prism reduces the review effort for identifying a Top-1 patch by over 32% (from 142 to 96) and simultaneously finds three additional Top-1 patches. In contrast, applying PATCH-SIM, ODS, and Shibboleth results in review effort reductions of 41%, 80%, and 11%, respectively, but each also decreases the Top-1 patch count by 7, 16, and 5. In total, Prism decreases the number of patches a developer must review for Top-1 patch identification by 12% (from 609 to 537) while increasing the Top-1 patch count by 9. Given that the maximum potential improvement in Top-1 patch detection is 37 (as indicated by the Top-∞ result), an increase of 9 represents a significant enhancement. In contrast, the other techniques reduce review effort by 20%, 84%, and 9% but lower the Top-1 patch count by 49, 118, and 24, respectively. The rule-based classifier prunes 641 incorrect patches—about twice as many as Prism, but results in the loss of 2, 9, and 13 Top-1, Top-5, and Top-∞ patches, respectively. While this performance drop is smaller than that of the existing techniques, it still highlights the challenge of preserving correct patches while filtering incorrect ones using hand-crafted rules. We discuss this issue further in Section 4.4.2. This key difference arises from our approach's precise patch representation using semantic features combined with a specialized learning algorithm tailored for APR scenarios. Note that, although Prism missed 7 correct patches, it did not degrade the end-to-end APR performance. We will discuss this further in Section 4.4.3.

4.2 RQ2: Patch Classification Performance

In this experiment, we directly compare the patch classification performance of Prism and other APCC techniques, without considering ranking aspects. This evaluation approach is standard in prior work [10, 24, 45–47, 52, 56].

Evaluation Settings. We evaluated the patch classification performance of each APCC technique using our previously described patch classification dataset as the benchmark. Our evaluation focused on two key metrics: the correct patch preserving rate (CPR) and the incorrect patch detection rate (IDR). These metrics typically exhibit a trade-off, where improvements in one often lead to a decline in the other. Because each APCC technique employs a different threshold for patch classification, we systematically varied these thresholds to evaluate their performance under uniform conditions

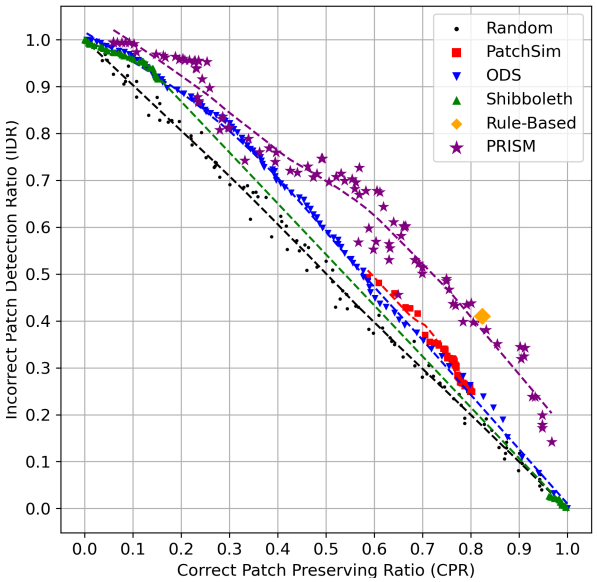


Fig. 5. Trade-off between CPR and IDR across varying thresholds. Each point (x, y) represents the CPR (x-axis) and IDR (y-axis) of a technique at a specific threshold setting.

and to examine their impact on the CPR/IDR balance. For example, PATCH-SIM provides a tunable threshold K_p, which controls its sensitivity to changes in execution traces of passing tests; we varied K_p from 0.01 to 1. As ODS and Shibboleth do not offer an explicit threshold, we adjusted the decision threshold of their binary classifiers (XGBoost [3] and Random Forest [11], respectively), which is set to 0.5 by default. Since Rule-Based uses a fixed rule without a tunable threshold, we only report a single CPR/IDR point. For Prism, we varied the CPR threshold γ, of our learning algorithm over the same range (0.01 to 1).

Results. Figure 5 shows that Prism achieves the best balance between CPR and IDR, delivering superior incorrect patch detection performance without sacrificing CPR. Across nearly all CPR ranges, Prism consistently maintains a higher IDR than the other techniques. Rule-Based also shows competitive performance (CPR: 82%, IDR: 41%), which is expected as it represents a subset of Prism's learned formula. The result indicates that, for a given level of correct patch preservation, our approach is able to eliminate more incorrect patches, thereby reducing the human effort required for patch validation. Notably, in the high-CPR range (above 50%), which is critical for APCC in APR contexts, Prism outperforms the competing methods by a substantial margin.

4.3 RQ3: Ablation Study

In this section, we examine the contributions of the key components of Prism: its semantic features and learning algorithm, to its overall performance.

Evaluation Settings. In this experiment, we used the same patch classification dataset as in Section 4.2 to assess the contributions of different feature sets and the learning algorithm. To evaluate feature importance, we compared three configurations: (1) using our proposed 66 semantic features (Sem), (2) using the 370 syntactic features (Syn) proposed by Ye et al. [56] (the original paper reports 202 features, but the published implementation actually uses 370), and (3) using a combination

Table 6. Ablation study on IDR across various feature types: syntactic feature (Syn), semantic feature (Sem), and both features (Syn+Sem), and models: random forest (RF) and ours (Prism) with fixed CPR (70%, 80%, and 90%). The best results for each setting are highlighted in **bold**.

Feature Type	IDR at CPR=70%		IDR at CPR=80%		IDR at CPR=90%	
	RF	Prism	RF	Prism	RF	Prism
Syn	638 (48%)	650 (50%)	405 (30%)	375 (28%)	265 (20%)	135 (10%)
Sem	623 (47%)	672 (51%)	484 (37%)	**572 (43%)**	339 (25%)	**451 (34%)**
Syn+Sem	649 (49%)	**723 (55%)**	422 (32%)	544 (41%)	167 (12%)	340 (26%)

of both feature sets (Syn + Sem). To assess the significance of our learning algorithm (Prism), we compared its performance against a baseline Random Forest classifier (RF). All evaluations were conducted under identical conditions by comparing the highest IDR values achieved at fixed CPR thresholds of 70%, 80%, and 90%.

Results. Table 6 demonstrates that the combination of our semantic features with our specialized learning algorithm is essential for achieving a high incorrect patch filtering rate without sacrificing correct patches. In high-CPR regions (80% and 90%), our approach using semantic features alone records the highest IDR values, 43% at CPR=80% and 34% at CPR=90%. At a CPR of 70%, the combination of our learning algorithm with combined feature sets produces superior results. We observed that at lower CPR thresholds our algorithm generally produces a smaller formula, thereby requiring a relatively smaller search space; consequently, enriching the representation with diverse features proves beneficial. Notably, our 66 semantic features consistently outperform 370 syntactic features by achieving a higher IDR across all CPR levels. This demonstrates that our compact but more expressive semantic features can represent each patch more effectively than conventional syntactic features, which enables our learning algorithm to train a more effective patch classifier in less time by working with fewer features. In fact, in high-CPR regions, where more complex formulas are typically required, the performance achieved using only semantic features even exceeds that of the combined feature set.

4.4 Discussion

In this section, we discuss (1) the cost of Prism in capturing semantic features from a signle patch, (2) the characteristics of the incorrect patches identified by Prism's learning algorithm, (3) the cases where Prism mistakenly classifies correct patches as incorrect, and (4) practical guidance for adopting Prism in APR systems.

4.4.1 Cost Analysis of Prism. We conducted a cost analysis for all benchmarks with Prism by recording the elapsed time for three steps: (1) the compilation (capture) process, (2) the execution of developer-provided tests to extract runtime information, and (3) the main analysis for the original program and patched program. The average times for these steps were 22, 102, and 54 seconds, respectively. The first two steps (compilation and test execution) can typically be integrated into the plausible patch validation process in general APR systems. The main analysis, which is the essence of Prism, does not cause significant overhead due to its scalable analysis design. For the learning algorithm, we allocated 3 minutes to train the model. Although this is slower than off-the-shelf methods (e.g., Random Forest), training is performed only once and the model can be reused for multiple repairs. Therefore, we believe that a 3-minute training time is a reasonable trade-off, considering the substantial long-term benefits it provides for APR.

4.4.2 Characteristics of Incorrect Patches. Most patches successfully classified as incorrect by Prism belong into two major categories: patches (1) causing unexpected crashes, and (2) eliminating the developer's original intent. Since the former is well-discussed in prior researches [9, 40, 55], we focus on the latter with examples. We note that these observations were possible because our learning algorithm uses an interpretable classification model based on boolean formulas.

Making Originally Used Variables Unused. In 37% of patches that were classified as incorrect, previously used variables made unused. Because developers basically declare variables only when they intend to use them, making variables dead often contradicts the developer's intent. Consider the following incorrect patch for bug Chart-5 by generated by Avatar [26]:

```
1 public XYDataItem addOrUpdate(double x, double y) {
2 -  return addOrUpdate(new Double(x), new Double(y));
3 +  return addOrUpdate(new Double(y), new Double(y));
4 }
```

This patch replaces `x` with `y` on line 2, removing all uses of the parameter `x` within the method.

Making Methods Always Return the Same Value. 19% of incorrect patches identified by Prism made a method always return the same value, harming its functionality. For example, consider the following incorrect patch for Lang-63 generated by Avatar:

```
1 static int reduceAndCorrect(...) {
2     if(endValue < startValue){
3 -       int newdiff = startValue - endValue;
4 +       int newdiff = endValue - endValue;
5         return newdiff;
6     } else return 0;
7 }
```

Originally, it computes the difference between `startValue` and `endValue`. However, the patch eliminates the original functionality by making the method always return 0.

Removing Developer-Intended Exceptions. 18% of patches, identified as incorrect, removed exceptions intended by developers to handle specific error conditions. For example, consider the following incorrect patch by TBar for Math-85:

```
1 static double[] bracket(...) throws ConvergenceException {
2     ...
3     if (fa * fb >= 0.0)
4 -       throw new ConvergenceException(...);
```

It is defined to throw a `ConvergenceException` in its signature. However, the patch made the `ConvergenceException` never occur, contradicting the developer's intent.

These anti-patterns inspired the construction of a rule-based classifier (Rule-Based) used in our evaluation. However, as shown in the experimental results, building an effective rule-based classifier is not only non-trivial but also suboptimal. One key challenge lies in combining multiple features without unintentionally filtering out correct patches. Our learning algorithm addresses this by automatically exploring a large space of boolean formulas. We observed that some correct patches exhibit "suspicious" features commonly found in incorrect patches, but are still valid due to compensating features. For example, in Section 2, the formula includes a seemingly harmful feature

"Do not eliminate array with IOB" yet the patch is correctly accepted based on other supporting evidence. Such reasoning is difficult to encode manually.

Importantly, while these anti-patterns do not definitively indicate incorrectness, we believe these insights offer valuable guidance for future APR and APCC research. For instance, discouraging the generation of patches with such features or assigning them lower rankings may further improve repair performance and increase the likelihood of identifying correct patches.

4.4.3 Case Analysis for Misclassification of Correct Patches. Our experiments show that Prism does not perfectly preserve all correct patches, as features common in incorrect patches sometimes appear in correct ones. Among the various types of misclassifications we observed, we focus on two particularly interesting cases: (1) misclassification due to imprecise static analysis and (2) misclassifications that have negligible impact on overall APR performance.

Misclassification due to Imprecise Analysis. Since Prism extracts semantic features based on the results of static analysis, imprecision in static analysis can lead to spurious features. For example, consider the following code snippet from the developer's fix for bug Math-71:

```
1 while (!lastStep) {
2   for (int k = 1; k < stages; ++k)
3     for (int j = 0; j < y0.length; ++j) {
4   ...
5   for (boolean loop = true; loop;) {
6     stepSize = hNew;
7     if (...)
8 +     hNew = 0;
```

In this patch, the variable `hNew` is re-initialized at line 8 and used in the loop at line 6. However, Prism failed to accurately analyze the complex loop structure, misidentifying `hNew` as a dead variable. As a result, this patch was misclassified as incorrect.

Misclassification with Minimal Impact on APR. In Section 4.1, despite missing some correct patches, Prism did not decrease the repair capability (Top-∞) of APR. For example, Prism filtered out the following correct patch generated by TBar for the Closure-126 bug:

```
1 if (NodeUtil.hasFinally(n)) {
2    Node finallyBlock = n.getLastChild();
3 -  tryMinimizeExits(finallyBlock, exitType, labelName);
4 }
```

It is correct but misclassified due to generation of a dead variable, `finallyBlock`, on line 2. However, APR tools typically generate multiple patches for a bug and can easily generate patches without such unused variables. For instance, TBar also generated the following patch for the same bug, which does not contain dead variables:

```
1 - if (NodeUtil.hasFinally(n)) {
2 -   Node finallyBlock = n.getLastChild();
3 -   tryMinimizeExits(finallyBlock, exitType, labelName);
4 - }
```

This patch was not filtered by Prism, ensuring that the final patch performance is not degraded.

4.4.4 Practical Guidance for Adopting Prism in APR. A common concern when using Prism is the need to retrain the model with newly labeled patches. However, as discussed earlier (Section 4.4.2), the models trained on different projects consistently learned similar key semantic features, and achieved comparable performance across datasets. On average, our six project-specific models preserved 91.2% of correct patches (standard deviation: 1.05%) while filtering out 40.3% of incorrect ones (standard deviation: 1.61%). These results demonstrate the robustness and consistency of Prism across diverse codebases, suggesting that Prism can be immediately deployed in practical APR systems with minimal setup effort, such as directly using our provided models. Furthermore, for scenarios where Prism is applied continuously to a specific codebase, we recommend collecting a small number of labeled patches from that project. This lightweight transfer learning step allows users to fine-tune the model for improved accuracy and better alignment with project-specific characteristics, without requiring a full retraining dataset.

5 Related Work

Automated Patch Correctness Classification. To address testcase overfitting in automated program repair [21, 39, 41], various methods for automatic patch correctness classification (APCC) have been proposed [7, 10, 13, 24, 43, 45, 46, 52, 55, 56, 61]. These approaches can be broadly categorized into two groups: (1) dynamic execution-based methods and (2) syntax-based methods. Dynamic execution-based approaches [13, 52, 55] leverage runtime information to classify patch correctness. For instance, Opad [55] identifies incorrect patches by checking for violations of memory-safety and crash oracles. PATCH-SIM [52] compares execution traces under the assumption that significant changes in passing test traces indicate an incorrect patch. Poracle [13] enhances the test oracle with user-provided preservation conditions in a semi-automated fashion. However, these methods are limited by the incomplete coverage of test executions and their high time consumption, which restricts their scalability in real-world APR scenarios. Syntax-based methods [24, 43, 45, 46, 56] focus on analyzing code structure and patterns introduced by patches. Tan et al. [43] proposed seven syntactic anti-patterns and considered patches incorrect if they exhibit such features. Recent approaches represent patches using various syntactic features and classify them with learning models. These methods employ either a hand-crafted set of features [56], learned embeddings [24, 45], or a combination of both [46]. Compared to dynamic techniques, syntax-based approaches are faster as they do not require program execution. However, they may fail to capture subtle semantic changes that can significantly impact patch correctness. Additionally, hybrid approaches such as Shibboleth [10] combine static and dynamic features. This method is based on the assumption that correct patches tend to induce minimal changes in both syntax and execution traces, and should not reduce the code coverage of passing tests. Yet, even these balanced designs can overlook minor semantic variations that are critical for patch correctness. With recent advances in large language models (LLMs), Zhou et al. [61] leverage LLMs for patch classification, demonstrating that effective patch classification is achievable without fine-tuning or a large labeled dataset. Although all the existing works have demonstrated high classification accuracy on datasets, they have not been evaluated for their usability in real-world APR scenarios. In this paper, we propose Prism to addresses this critical gap by directly targeting the unique challenges encountered in practical APR.

Automated Patch Correctness Assessment. APR techniques are primarily evaluated subjectively by their authors, which can sometimes lead to unreliable results [18]. To address this issue, recent studies have focused on automatically assessing patch correctness (APCA) [22, 47, 51, 54, 58]. Unlike APCC methods, APCAs generally assume an ideal oracle, which is not practical in real-world scenarios. One major approach in APCA is to identify tests that produce different results between

the patched program and the ground truth. For instance, DiffTGen [51] uses EvoSuite [8] to generate tests targeting the patched areas for detecting behavioral differences. Ye et al. [58] conducted a large-scale experiment evaluating the correctness of 638 patches using random testing tools [8, 38]. Another method compares runtime states during execution. Yang and Yang [54] evaluated patches based on runtime invariants extracted by Daikon [6]. Invalidator [22] leverages both runtime invariants and syntactic patterns of correct patches to assess patch correctness. A significant large-scale study by Wang et al. [47] evaluated 902 patches from 21 APR tools on Defects4J bugs, comparing the performance of APCA and APCC techniques.

Other Techniques to Improve APR. To enhance APR performance, various methods have been proposed, such as extracting patch ingredients by solving semantic constraints [19, 20, 33, 34, 36, 60], prioritizing patches likely to be correct based on characteristics observed in developer patches [1, 26, 28–30, 48], preventing the generation of crash-inducing patches [9, 40], and enhancing test oracles [2, 14, 44]. Recently, large language models (LLMs) have also been used to improve patch generation in APR [50, 53]. These LLM-based approaches demonstrate the ability to rewrite entire functions, not just small code fragments [53]. We believe that detecting overfitting is more challenging in such rewrite-based APR, where patches often introduce substantial syntactic and structural modifications. In these cases, existing syntactic or trace-based APCC techniques [10, 24, 43, 45, 46, 52, 56] may become less effective due to the extent of the changes. Prism is particularly well-suited for these scenarios, as it focuses on the underlying behavioral changes. APCC techniques, including Prism, can also be considered as part of these approaches since they reduce the review effort by pruning incorrect patches and increasing the likelihood of identifying correct ones. However, for APCC techniques to be useful in APR, it is crucial that they preserve as many correct patches as possible while effectively filtering out incorrect ones. Experimentally, we found that Prism is the only technique that consistently satisfied this requirement, demonstrating its practical benefit. Furthermore, we believe that the key features discovered by Prism (Section 4.4.2) can help the design of future APR techniques.

6 Conclusion

We presented Prism, a novel APCC technique that enhances real-world APR systems by accurately identifying overfitting patches. Unlike existing methods that often misclassify rare correct patches, reducing repair effectiveness, our approach leverages new semantic features and a specialized learning algorithm to create a patch classifier optimized for APR. Evaluations with 10 APR tools show that Prism improves repair performance while reducing human review effort. Furthermore, experiments on a large dataset of 1,829 labeled patches demonstrate that, at equivalent correct patch preservation rates, Prism surpasses prior techniques in filtering out incorrect patches. We believe our work sets a clear path for future APCC research and its practical use in real APR scenarios.

Acknowledgments

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair, No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains, 5%) and by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2025-RS-2020-II201819, 5%). This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2021R1A5A1021944, 20%).

Data Availability

The source code of our tool implementation, benchmarks we used, and experimental results are archived on Zenodo at [42] and also available at https://github.com/PRISM-artifact/PRISM.

References

[1] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. doi:10.1145/3360585

[2] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. 2020. Human-In-The-Loop Automatic Program Repair. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 274–285. doi:10.1109/ICST46399.2020.00036

[3] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. doi:10.1145/2939672.2939785

[4] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* 47, 09 (sep 2021), 1943–1959. doi:10.1109/TSE.2019.2940179

[5] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J. Hellendoorn. 2021. Patching as translation: the data and the metaphor. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 275–286. doi:10.1145/3324884.3416587

[6] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123. doi:10.1109/32.908957

[7] Zhiwei Fei, Jidong Ge, Chuanyi Li, Tianqi Wang, Yuning Li, Haodong Zhang, LiGuo Huang, and Bin Luo. 2025. Patch Correctness Assessment: A Survey. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 55 (Jan. 2025), 50 pages. doi:10.1145/3702972

[8] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. doi:10.1145/2025113.2025179

[9] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 8–18. doi:10.1145/3293882.3330558

[10] Ali Ghanbari and Andrian Marcus. 2022. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Virtual</city>, <country>South Korea</country>, </conf-loc>) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 654–665. doi:10.1145/3533767.3534368

[11] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Vol. 1. 278–282 vol.1. doi:10.1109/ICDAR.1995.598994

[12] Facebook Inc. 2018. A tool to detect bugs in Java and C/C++/Objective-C code before it ships. Available: https://fbinfer.com.

[13] Elkhan Ismayilzada, Md Mazba Ur Rahman, Dongsun Kim, and Jooyong Yi. 2023. Poracle: Testing Patches under Preservation Conditions to Combat the Overfitting Problem of Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 44 (dec 2023), 39 pages. doi:10.1145/3625293

[14] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2018. OASIs: oracle assessment and improvement tool. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 368–371. doi:10.1145/3213846.3229503

[15] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. doi:10.1145/3213846.3213871

[16] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1161–1173. doi:10.1109/ICSE43902.2021.00107

[17] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco,

CA, USA) *(ICSE '13)*. IEEE Press, 802–811.

[18] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On Reliability of Patch Correctness Assessment. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 524–535. doi:10.1109/ICSE.2019.00064

[19] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 376–379. doi:10.1145/3092703.3098225

[20] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 593–604. doi:10.1145/3106237.3106309

[21] Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in Semantics-Based Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 163. doi:10.1145/3180155.3182536

[22] Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D. Le, David Lo, Nhat-Hoa Tran, Bui Quang-Huy, and Quyet-Thang Huynh. 2023. Invalidator: Automated Patch Correctness Assessment Via Semantic and Syntactic Reasoning. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3411–3429. doi:10.1109/TSE.2023.3255177

[23] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. doi:10.1109/TSE.2011.104

[24] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-Aware Code Change Embedding for Better Patch Correctness Assessment. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 51 (may 2022), 29 pages. doi:10.1145/3505247

[25] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42. doi:10.1145/3293882.3330577

[26] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 456–467.

[27] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 615–627. doi:10.1145/3377811.3380338

[28] Kui Liu, Jingtang Zhang, Li Li, Anil Koyuncu, Dongsun Kim, Chunpeng Ge, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Reliable Fix Patterns Inferred from Static Checkers for Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 96 (may 2023), 38 pages. doi:10.1145/3579637

[29] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727–739. doi:10.1145/3106237.3106253

[30] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 298–312. doi:10.1145/2837614.2837617

[31] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. doi:10.1145/3395363.3397369

[32] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 441–444. doi:10.1145/2931037.2948705

[33] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 448–458. doi:10.1109/ICSE.2015.63

[34] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 691–701. doi:10.1145/2884781.2884807

[35] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based Neural Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1456–1468. doi:10.1109/ICSE48619.2023.00127

[36] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781. http://dl.acm.org/citation.cfm?id=2486788.2486890

[37] Yicheng Ouyang, Jun Yang, and Lingming Zhang. 2024. Benchmarking Automated Program Repair: An Extensive Study on Both Real-World and Artificial Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 440–452. doi:10.1145/3650212.3652140

[38] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 815–816. doi:10.1145/1297846.1297902

[39] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 24–36. doi:10.1145/2771783.2771791

[40] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 390–405. doi:10.1145/3453483.3454051

[41] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543. doi:10.1145/2786805.2786825

[42] Dowon Song and Hakjoo Oh. 2025. *PRISM Artifact: Enhancing APR with PRISM*. doi:10.5281/zenodo.16899847

[43] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-Patterns in Search-Based Program Repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 727–738. doi:10.1145/2950290.2950295

[44] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary improvement of assertion oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1178–1189. doi:10.1145/3368089.3409758

[45] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2021. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 981–992. doi:10.1145/3324884.3416532

[46] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F. Bissyandé. 2023. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 92 (may 2023), 34 pages. doi:10.1145/3576039

[47] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2021. Automated Patch Correctness Assessment: How Far Are We?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 968–980. doi:10.1145/3324884.3416590

[48] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3180155.3180233

[49] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971. doi:10.1145/3540250.3549101

[50] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software*

Testing and Analysis (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 819–831. doi:10.1145/3650212.3680323

[51] Qi Xin and Steven P. Reiss. 2017. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 226–236. doi:10.1145/3092703.3092718

[52] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 789–799. doi:10.1145/3180155.3180182

[53] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2025. Aligning the Objective of LLM-Based Program Repair . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2548–2560. doi:10.1109/ICSE55347.2025.00169

[54] Bo Yang and Jinqiu Yang. 2020. Exploring the Differences between Plausible and Correct Patches at Fine-Grained Level. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. 1–8. doi:10.1109/IBF50092.2020.9034821

[55] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 831–841. doi:10.1145/3106237.3106274

[56] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2022. Automated Classification of Overfitting Patches With Statically Extracted Code Features. *IEEE Transactions on Software Engineering* 48, 8 (2022), 2920–2938. doi:10.1109/TSE.2021.3071750

[57] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2023. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 92, 13 pages. doi:10.1145/3551349.3556926

[58] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated Patch Assessment for Program Repair at Scale. *Empirical Softw. Engg.* 26, 2 (mar 2021), 38 pages. doi:10.1007/s10664-020-09920-w

[59] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. doi:10.1145/3510003.3510222

[60] Jooyong Yi and Elkhan Ismayilzada. 2022. Speeding up Constraint-Based Program Repair Using a Search-Based Technique. *Inf. Softw. Technol.* 146, C (jun 2022), 17 pages. doi:10.1016/j.infsof.2022.106865

[61] Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. 2024. Leveraging Large Language Model for Automatic Patch Correctness Assessment. *IEEE Transactions on Software Engineering* 50, 11 (2024), 2865–2883. doi:10.1109/TSE.2024.3452252

[62] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 341–353. doi:10.1145/3468264.3468544

Received 2025-03-26; accepted 2025-08-12