Which Inputs Trigger my Patch?

Martin Eberlein
Humboldt-Universität zu Berlin
Berlin, Germany
martin.eberlein@hu-berlin.de

Moeketsi Raselimo
Humboldt-Universität zu Berlin
Berlin, Germany
raselimm@informatik.hu-berlin.de

Lars Grunske
Humboldt-Universität zu Berlin
Berlin, Germany
grunske@informatik.hu-berlin.de

***Abstract*—Automated program repair (APR) has made significant progress in autonomously locating and fixing software faults. However, developers face uncertainty in trusting and validating patches, as the boundaries of the patch and the concrete inputs the patch fixes often remain unclear.**

In this paper, we introduce AVICENNAPATCH, a novel tool designed to provide precise explanations of the input structures that are covered by the patch. Building on the AVICENNA framework, AVICENNAPATCH uses a differential testing strategy, executing both the original and patched program versions to identify inputs where behavior diverges. By leveraging ISLA constraints, AVICENNAPATCH generates human-readable, grammar-based explanations that characterize patch-triggering inputs in terms of syntactic and semantic features. To refine these explanations, AVICENNAPATCH iteratively generates new inputs and adjusts its hypotheses, ensuring that the explanations are both precise and generalizable. Our approach not only enhances transparency in APR by clarifying the operational boundaries of patches but also supports developers in evaluating the reliability and robustness of automated fixes across diverse input scenarios. Our evaluation on 12 patches for 7 different subjects shows that AVICENNAPATCH can provide effective and general explanations.

***Index Terms*—debugging, testing, patch validation**

I. INTRODUCTION

In automated program repair (APR), an ongoing challenge is to ensure that a generated patch is both general and effective in the scenarios it aims to address. While recent advances in APR have significantly improved the ability to detect and repair faults automatically, understanding the conditions under which these patches are activated—which inputs trigger the patch—often remains unclear. Specifically, when a patch is generated to address a particular issue in code, there is often little insight into what kinds of inputs are covered by its execution. This lack of transparency can hinder a developer's ability to trust and validate automatically generated patches, as the boundaries of their effectiveness remain opaque.

Understanding when a patch gets triggered is vital for numerous reasons. Developers need to know the types of inputs and runtime conditions that will exercise the patched code, particularly to avoid situations where a patch may introduce unexpected behavior. For example, if a patch is triggered only under rare input conditions, this might indicate that it solves a highly specialized corner case rather than addressing the broader fault. On the other hand, if it triggers too broadly, there is a risk of the patch negatively impacting other, unrelated functionalities. Knowing the precise conditions under which a patch is activated can thus play a pivotal role in enhancing the reliability and predictability of APR.

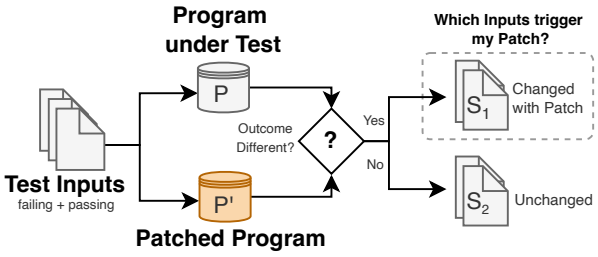

Fig. 1. Differential Behavior Testing. We use a differential oracle to detect whether a patch changes the outcome of a test input. AVICENNAPATCH then automatically identifies the characteristics of the inputs affected by the patch.

In this paper, we introduce AVICENNAPATCH, a novel approach to explain the specific input scenarios that activate a patch, filling a critical gap in APR transparency. We use a differential oracle to determine which inputs are affected by the patch (Figure 1). This process begins with a set of initial test inputs, which include both passing and failing cases that utilize the patch in question. These inputs are executed on both the original program P and the patched program P'. By comparing outcomes between P and P', we identify inputs where behavior differs as a result of the patch. Our objective is to derive a precise and generalized explanation that describes all inputs for which the outcome has changed (S_1), while excluding cases where the patch does not affect behavior (S_2).

To systematically analyze and describe these differences, we parse the inputs with a provided input specification (i.e., a grammar) that captures key syntactic and semantic features. This allows us to pinpoint common input characteristics associated with the patch's activation. We then use these features to automatically generate a detailed explanation of the conditions under which the patch is triggered. These explanations are represented as ISLA constraints [1], a domain-specific language for specifying and analyzing constraints on structured inputs.

Since initial explanations may be inaccurate or overly specific, we employ a hypothesis refinement loop, iteratively adjusting or confirming our explanations to generalize them effectively. Through this process, we provide developers with actionable insights into the precise input conditions for patch effectiveness, aiding in evaluating and validating the patch's applicability and reliability.

```python
import math

def logarithm(x: float):
    return math.log(x)
```

Fig. 2. A simple function computing the natural logarithm of a number x. While straightforward, it does not handle edge cases such as negative values, leading to errors.

Example. To illustrate how AVICENNAPATCH works, consider the example in Figure 2, which shows a simple Python function for computing the natural logarithm of a given value x. However, it does not handle edge cases such as $x = 0$ or negative values, causing errors and incorrect behavior: calling `logarithm(0)` raises an exception because the logarithm is undefined at zero, while calling `logarithm(-34)` raises an exception because the logarithm of a negative number is undefined in the real domain.

A language model like ChatGPT [2] can easily generate a patched version of this function, as shown in Figure 3. This revised implementation addresses the problematic edge cases with explicit checks: (i) it returns `-inf` when $x = 0$, avoiding an error, and (ii) ensures x is both numeric and positive; otherwise, a `TypeError` is raised.

At first glance, this patch may appear straightforward, but it can be deceptively difficult to determine exactly which inputs it handles. Developers must carefully analyze how each function component behaves under a range of inputs, especially in scenarios involving nested or subtle conditions. AVICENNAPATCH helps address these challenges by systematically generating constraint-based explanations for inputs that activate specific conditions. Using a defined input format specification (e.g., a grammar), AVICENNAPATCH constructs an explanations as ISLA constraints, clarifying which input forms the patch covers:

```
exists ⟨float⟩ number in start:
    (number = "0") or
exists ⟨maybe-minus⟩ sign in ⟨float⟩:
    (sign = "-")
```

Given this explanation, the following input cases are identified as covered by the patch:

Floating-point numbers equal to zero: When the ⟨*float*⟩ element in the input is exactly zero, the patch handles this case by returning a controlled response (e.g., `-inf`) instead of causing an error as in the unpatched version.

Negative floating-point numbers: The explanation shows that when the ⟨*maybe-minus*⟩ element within ⟨*float*⟩ matches a negative sign, the patch correctly identifies negative floating-point numbers and handles values less than zero appropriately. This prevents the function from attempting to compute the logarithm of a negative number, thereby avoiding the uncontrolled exception in the original implementation.

By analyzing these input characteristics, AVICENNAPATCH provides a precise mapping between input structures and the patched code's behavior. This helps developers understand why certain inputs trigger the patch and confirms that edge cases are correctly addressed.

```python
import math

def logarithm(x: float):
    # Patch applied to cover edge cases
    if x == 0:
        return float("-inf")
    elif x > 0:
        return math.log(x)
    else:
        raise TypeError("Input must be a" \
                        "positive value")
```

Fig. 3. Patched logarithmic function. It explicitly checks for zero and negative numbers. These changes ensure robust handling of edge cases.

While this specific patch is straightforward, real-world patches may involve intricate conditions that are challenging to track manually. By systematically explaining which inputs activate a patch, AVICENNAPATCH increases transparency and confidence in automatically generated patches. This clarity is crucial for ensuring patches are robust, especially in complex codebases with intricate input interactions. Moreover, our approach extends beyond just explaining a single patch. It systematically identifies and categorizes the inputs affected by any patch, providing explanations that are both interpretable and applicable to a wide range of patches. This enables developers to:

Validate Patch Coverage. Confirm that the patch correctly handles all intended input scenarios without leaving gaps or introducing unintended side effects.

Improve Testing. Use solvers, like ISLA, and the derived explanations to generate targeted test inputs that thoroughly exercise the patched behavior.

Enhance Debugging. Quickly identify which inputs cause unexpected patched behavior, facilitating faster diagnosis of complex issues.

By helping developers understand how patches change program behavior for different inputs, AVICENNAPATCH provides insights needed to ensure patches work as intended and improve software reliability. With clear, automated explanations of patch-triggering conditions, developers gain the confidence to maintain and evolve patched code effectively.

In summary, we make the following contributions:

Patch Input Explanation. We present AVICENNAPATCH, a novel framework designed for generating descriptive explanations that summarize the types of inputs affected by a patch. These explanations clarify the operational boundaries of the patch, enabling developers to assess its behavior across diverse input scenarios.

Evaluation. We evaluate AVICENNAPATCH on 12 different patches, demonstrating that it can accurately approximate the input space affected by each patch. Our results show that we generate precise explanations, effectively predicting and producing inputs that trigger the patches.

The remainder of this paper is organized as follows. Section II discusses the related work as well as the concepts and frameworks that our approach leverages. In Section III, we detail the principles of AVICENNAPATCH, followed by its implementation in Section IV. Section V evaluates AVICENNAPATCH on a number of patches, assesing the precision and accuracy of the explanations. Section VII closes with conclusion and future work.

II. BACKGROUND AND RELATED WORK

A. Patch Testing

Patch testing can be used to demonstrate the presence of bugs in the generated patch (or fix) for a program under test. Existing work [3], [4], [5] has documented the extent to which patches provided to several large scale software are buggy. We therefore draw related work from two competing techniques (i.e., directed symbolic execution and directed greybox fuzzing) that have been applied to test patches.

Directed symbolic execution [6], [7], [5], [8], [9] guides the analysis to specific parts of the program, e.g., a target line, using various heuristics such as maximizing a certain fitness function [6] or following a path along a given control-flow graph (CFG) [7]. These approaches, in principle, can be more efficient and precise than AVICENNAPATCH, however, their inherent reliance on heavy program analysis comes with scalability issues.

Greybox fuzzing on the other hand has seen recent advancements, and researchers have taken advantage of its strengths to address the aforementioned scalability issues of symbolic execution based approaches. These fuzzing strategies in their basic form perform a biased and random search over program input space, and directed greybox fuzzing [10], [11], [12], [13], [14] lifts this idea and tries to guide exploration towards certain locations in the program under test. This idea lends itself readily for testing patches–use the patched lines as the target for fuzzing. In deed, the authors of AFLGo [10] show how the symbolic execution based tool Katch [5] fails to reproduce a vulnerability within 24 hours, while AFLGo reproduces it within minutes. Other state-of-the-art directed greybox fuzzing tools that have been used for patch testing include: Beacon [11], SelectFuzz [12], and DAFL [13]. However, directed greybox fuzzing is only a testing method and does not give any explanations to program failures. Exploring the synergy between fuzzing and AVICENNAPATCH is an interesting direction for future work.

B. Automatic Program Repair

Many methods have been applied to the problem of automatic program repair (APR). Starting with Forrest et al. [15]'s seminal work on GenProg, a variety of different search-based approaches and algorithms have been investigated, ranging from genetic programming techniques (GenProg family of approaches) [15], [16], [17], [18], [19] to mutation-based approaches [20], [21], [22] and those that exploit templates [23], [24], [25]. Techniques in this camp are also known as generate-and-validate techniques, in which patches are generated via program edit operations and validated dynamically, typically over a given test suite.

Another line of work uses symbolic analysis to execute faulty lines to derive repair constraints and exploit component-based program synthesis to solve the repair constraints [26], [27], [28].

More recent work takes advantage of advances in machine learning. Here, we can further classify approaches into two categories: (i) Neural Machine Translation (NMT) based techniques, which essentially reduce an APR task into an NMT task and translates buggy code into correct code. Notable tools in this category include TENURE [29], Tare [30], SelfAPR [31], RewardRepair [32], Recoder [33], CURE [34], and CoCoNuT [35]. And (ii) Large Language Models (LLMs), which have also been subjected to the problem of APR and have been shown to outperform all existing approaches [36]. While they differ in their capabilities, LLM-based APR tools typically take an input prompt and the buggy code and query the underlying LLM to generate a patch at a given location. This basic idea was first realized in the tool AlphaRepair [37]. FitRepair [36] improves on AlphaRepair and incorporates the *plastic surgery hypothesis* in which existing, correct code fragments are reused to repair buggy code fragments.

Despite these advancements, overfitting remains the key challenge in APR techniques [38]. Several approaches have been proposed to combat the overfitting problem in APR: Gao et al. [39] proposes ExtractFix that addresses this challenge via symbolic reasoning. More specifically, given a crash location, ExtractFix first extracts a *crash-free constraint*, i.e., a property that all inputs should satisfy to avoid the vulnerability, and propagates this crash-free constraint to the fix location and finally patches are generated that ensure that the constraint is satisfied for all inputs. Another notable approach to address overfitting is implemented in the tool CPR [40]. The basic idea behind CPR is the co-exploration of both the patch and input spaces using concolic execution, and returns a ranked list of patches that are more precise. Moreover, Zhang et al. [41] describe a CEGIS-style approach that returns "likely patch invariants" mined using the Daikon system and leverages fuzz testing for dynamic tracing and refinement of the output. This approach outperforms both CPR and ExtractFix.

C. Semantic Debugging

Eberlein et al. introduced AVICENNA [42], a technique for automatically identifying failure causes using logical input properties. AVICENNA learns failure-constraints that describe the conditions under which a program fails, enabling effective fault isolation and debugging.

AVICENNAPATCH extends this idea to derive explanations for patches, focusing on their operational boundaries rather than faults. Unlike AVICENNA, which uses a fault-based oracle to classify inputs as fault-triggering or non-fault-triggering, AVICENNAPATCH employs a differential testing strategy. By comparing the outputs of original and patched program versions, AVICENNAPATCH identifies inputs that produce different outcomes, revealing how and when a patch affects program

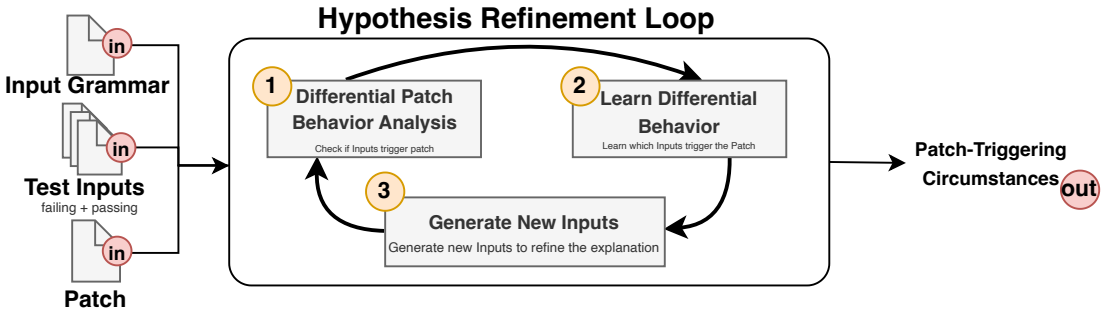

Fig. 4. Overview of AVICENNAPATCH's workflow. Starting with a grammar, initial test inputs, and a patch file, AVICENNAPATCH automatically generates an explanation detailing the characteristics of inputs affected by the patch.

behavior. This shift in focus expands AVICENNAPATCH's scope. While AVICENNA isolates fault-triggering inputs, AVICENNAPATCH classifies inputs based on patch-triggering behavior, offering insights into patch effectiveness, coverage, and configuration dependencies. Its differential oracle enable developers to understand the specific conditions under which patches are activated. In summary, AVICENNAPATCH builds on AVICENNA by providing patch-specific explanations through differential testing, enhancing transparency and reliability in automated program repair (APR).

III. APPROACH

In this section, we present the details of AVICENNAPATCH, an effective and general approach for constructing patch explanations. Our approach extends the AVICENNA framework to produce general and precise ISLA constraints, explaining the input conditions under which the program's behavior changes due to a patch. The key idea behind AVICENNAPATCH is to analyze how a patch changes a program's behavior on certain inputs. By identifying and examining these inputs—referred to as patch-triggering inputs—we aim to learn and refine constraints that characterize the new or changed program behavior. Through an iterative hypothesis refinement loop, we generate a precise explanation that captures all inputs affected by the patch. Figure 4 provides an overview of our tool's internal process.

Overview. Our approach begins with a program under test, a patch file, a set of initial inputs, and an input format specification in the form of a context-free grammar. To explain the patch, we require at least one input that triggers the patched behavior.

AVICENNAPATCH consists of three main steps to produce a comprehensive explanation of which inputs trigger the patch:

1) **Differential Patch Behavior Analysis (Section III-A)**: Determine which inputs cause different program behaviors between the original and patched versions.
2) **Learning Differential Behavior (Section III-B)**: Generate a general explanation describing input properties that trigger the patch, using the ISLearn [1] pattern-based constraint miner.
3) **Iterative Feedback Loop (Section III-C)**: Refine and validate these explanations by generating new inputs and observing program behavior.

This procedure allows us to associate the characteristics of inputs with observed changes in program behavior, ensuring that the explanations are both general and precise.

A. Differential Patch Behavior Analysis

AVICENNAPATCH focuses on identifying input properties that the patch covers rather than explaining failure-related circumstances like tools such as AVICENNA. To determine these properties, we use differential testing to analyze how the program's behavior changes after applying the patch.

The first step in AVICENNAPATCH involves determining whether the patch alters the program's behavior for given inputs. We use a *differential oracle* that compares outputs of the original program P and the patched version P' for the same inputs, as illustrated in Figure 1.

If the output differs between $P(x)$ and $P'(x)$ for an input x, we classify x as a *patch-triggering input* and include it in set S_1. If the output remains the same, x is considered a *non-triggering input* and placed in set S_2.

Formally, given an initial set of inputs I, we define:

$$S_1 = \{x \in I \mid P(x) \neq P'(x)\}$$

$$S_2 = \{x \in I \mid P(x) = P'(x)\}$$

Here, S_1 contains inputs that cause the patch to alter the program's behavior, while S_2 contains inputs where behavior remains unchanged. Identifying these sets allows us to focus on the properties of inputs in S_1, which are key to understanding the patch's impact.

B. Learning Differential Program Behavior

With the differential behavior established, the next step is to generalize the input characteristics that define S_1. Using a *pattern-based learning approach*, AVICENNAPATCH creates a semantic explanation that captures the properties of patch-triggering inputs.

Like AVICENNA, our approach relies on an input format specification provided as a context-free grammar G. This

grammar helps us associate syntactic features and semantic properties of inputs with observed program behavior changes. We employ ISLearn [1], a pattern-based constraint miner integrated into AVICENNA, to learn constraints that describe how the patch changes the program's behavior. This constraint learning process can be divided into two key steps:

Feature Extraction. For each input x in S_1 and S_2, we analyze its syntactic and semantic structure according to the grammar G. This analysis extracts features represented in the internal domain-specific language ISLA.

Constraint Synthesis. ISLearn uses these features to construct first-order logic formulas (ISLA constraints) that distinguish patch-triggering inputs in S_1 from non-triggering inputs in S_2. It employs a pattern-based approach to ensure that the learned constraints capture the essential characteristics of inputs triggering the patch while excluding those that do not.

Formally, the goal is the synthesize a constraint ψ such that:

$$\forall x \in S_1 : \psi(x) = \text{True}, \quad \forall y \in S_2 : \psi(y) = \text{False}$$

However, perfect separation of S_1 and S_2 may not always be achievable. In such cases, ψ should aim to maximize precision (accepting only inputs in S_1) and recall (capturing all inputs in S_1). This ensures ψ provides a meaningful explanation of the input conditions that activate the patch.

While ψ may generalize beyond the initial inputs, the initial explanation may not be precise enough, thus further refinement is necessary.

C. Iterative Feedback Loop

To ensure that the generated explanation (the ISLA constraint ψ) is both precise and general, AVICENNAPATCH employs an iterative hypothesis refinement loop. The refinement process consists of the following steps:

Input Generation. Based on the current hypothesis ψ, AVICENNAPATCH generates new inputs that should theoretically trigger the patch.

Testing and Validation. Each newly generated input is run on both the original program P and the patched program P'. The observed behavior helps in validating the correctness of ψ. If a new input satisfies ψ and triggers the patch as expected, it strengthens the confidence in the hypothesis. If a new input satisfies ψ but does not trigger the patch or if the patch triggers on an input not covered by ψ, it indicates that ψ needs refinement.

Constraint Refinement: Using the feedback from the newly generated inputs, ψ is refined to better capture the patch-triggering conditions. This may involve adjusting or combining constraints to ensure that all triggering inputs are included and non-triggering inputs are excluded.

This iterative process continues until ψ is both inclusive of all inputs in S_1 and exclusive of those in S_2, ensuring the final explanation accurately describes the patch-triggering conditions. The resulting refined ISLa constraint provides a robust and interpretable explanation of the patch's activation conditions. Developers can use this constraint to understand the impact and scope of the patch, ensuring more effective verification and maintenance of the program.

Summary. AVICENNAPATCH represents a paradigm shift from focusing on failure-related behaviors to analyzing how patches change program behavior. By automatically learning and refining constraints that capture the properties of inputs triggering the patch, AVICENNAPATCH extends AVICENNA's approach to provide developers with clear, human-readable explanations of patch effectiveness. This methodology supports more informed decision-making in patch development, application, and verification.

IV. IMPLEMENTATION

This section outlines the implementation of AVICENNAPATCH, which builds on the AVICENNA framework [42] and is implemented entirely in Python. We leverage AVICENNA's components, such as its integrated learner, using default parameters where applicable. While learning, the learned formulas are required to achieve at least a *Recall* of 90% and a *Precision* of 60%. These thresholds are designed to prioritize general explanations, ensuring the constraints are not overly specific while maintaining acceptable accuracy. Given a program and its corresponding patch file, AVICENNAPATCH automatically generates a differential oracle. This oracle is implemented as a single function that takes an input as a parameter and returns whether the input triggers the patch. It forms the foundation of the patch-triggering input classification, enabling subsequent learning and iterative refinement of constraints.

V. EVALUATION

We investigate the following research questions to assess how accurately our explanations approximate the input space of the patch:

RQ1 Predicting Patch-Triggered Inputs. How accurately can our explanations predict whether an input belongs to the input space affected by the patch? (Section V-B)

RQ2 Generating Patch-Triggered Inputs. How effectively can our explanations generate new inputs that belong to the input space affected by the patch? (Section V-C)

By evaluating the quality of our explanations both as predictors and producers, we aim to determine how well they approximate the patch's input space without overspecializing (which would make them accurate producers but inaccurate predictors) or overgeneralizing (which would make them accurate predictors but inaccurate producers).

A. Evaluation Setup

1) Subjects: To assess the effectiveness of AVICENNAPATCH, we evaluated our tool's diagnostic capabilities on a diverse set of subjects from the Tests4Py benchmark [43]. Our evaluation includes *12* patches drawn from *7* projects of varying size, functionality, and complexity, allowing us to explore AVICENNAPATCH's performance across different patch and program behaviors.

The subjects from Tests4Py include the following: *Logarithm* (used as a running example), *Calculator*, *Middle*, *Expression*, and *Markup*, as well as two real-world projects, *Pysnooper* [44] and *Cookiecutter* [45]. The latter two are considerably larger in size and complexity compared to the other subjects, containing significantly more lines of code and real-world functionalities, whereas the remaining subjects are simpler, toy examples containing only a few functions each [46]. Our selection covers a wide range of patch sizes, from small, localized changes of 1–3 lines to larger patches, such as in *Pysnooper.2*, which involves 35 lines of modifications. This variability in patch size enables us to evaluate AVICENNAPATCH's ability to handle patches with differing scopes and complexities. The Tests4Py benchmark extends BugsInPy [47] with functionality to verify inputs at the system level. This makes Tests4Py particularly suitable for evaluating AVICENNAPATCH, as it aligns with our goal of understanding the input conditions that trigger specific patches. Additionally, each subject in Tests4Py is accompanied by a grammar specifying the input format, allowing us to leverage this grammar for generating and analyzing patch-triggering inputs.

2) Data Sets: To answer RQ1, we require sets of test inputs to evaluate the prediction capabilities of AVICENNAPATCH's explanation. To generate these validation inputs, we use the k-path coverage guided, grammar-aware mutation fuzzer provided by ISLearn [1]. With the fuzzer, we automatically generate 200 unique validation inputs for each subject—100 patch triggering and 100 non-triggering test inputs. We measure the respective predictive power of the explanation based on these validation inputs. We evaluate AVICENNAPATCH's performance with an initial input corpus of two inputs only—one *patch-triggering* and one *non-triggering* input. This decision follows the idea that we want to know if our extended learning process and the feedback loop can generate meaningful additional inputs and thus improve its accuracy and precision. The two initial inputs, were provided by Tests4Py.

3) Research Protocol: To address RQ1 and RQ2, we followed a systematic procedure: (i) For each patch, we started AVICENNAPATCH using the respective grammar and the two initial inputs. These initial inputs serve as the starting point for the learning process. (ii) We performed 10 iterations of the learning and refinement process. This number is the default setting for the refinement loop in AVICENNAPATCH, and preliminary results indicated that it is sufficient for the explanations to converge. (iii) To account for randomness in the process, we repeated the experiments 10 times for each patch, using different random seeds to control variability.

For RQ1, we analyzed the predictive power of AVICENNAPATCH's final explanations for each patch. We then measured the performance of the explanations on the evaluation data set using *Precision* (the proportion of inputs predicted to trigger the patch that actually do so) and *Recall* (the proportion of all inputs that trigger the patch which are correctly predicted by the explanation). This assessment allowed us to evaluate how accurately the explanations could predict whether a new and unseen input would trigger the patch.

TABLE I
PRECISION AND RECALL WHEN USING AVICENNAPATCH AS A **PREDICTOR** AND **PRODUCER**. PRECISION AND RECALL ARE AVERAGES OVER 10 RUNS.

Subject	RQ1 Predictor		RQ2 Producer	
	Precision	*Recall*	*Precision*	*Recall*
Logarithm	100%	100%	100%	100%
Calculator	100%	100%	100%	100%
Middle.1	100%	100%	100%	100%
Middle.2	100%	100%	100%	100%
Expression	71%	93%	100%	66%
Markup.1	65%	100%	99%	100%
Markup.2	72%	100%	85%	100%
Pysnooper.1	100%	100%	100%	100%
Pysnooper.2	100%	100%	100%	100%
Cookiecutter.1	60%	92%	98%	76%
Cookiecutter.2	81%	93%	75%	89%
Cookiecutter.3	59%	100%	100%	100%
Total Average	**84%**	**98%**	**96%**	**94%**

For RQ2, we evaluated how effectively the generated explanations could produce new patch-triggering inputs. Using the ISLA solver, we generated 100 inputs predicted to trigger the patch and 100 inputs predicted not to trigger the patch based on the explanations. We then measured the reliability of the explanations as input generators by verifying, using *Precision* and *Recall*, whether the generated inputs indeed triggered or did not trigger the patch as expected.

B. RQ1: Predicting Patch-Triggered Inputs

To evaluate how accurately AVICENNAPATCH's explanations approximate the input space affected by the patch, we measured the *Precision* and *Recall* of the final explanations on the evaluation dataset. The results are summarized in Table I under the **Predictor** columns.

For several subjects, AVICENNAPATCH achieved perfect approximation of the patch's input space. Specifically, for *Logarithm*, *Calculator*, *Middle.1*, *Middle.2*, *Pysnooper.1*, and *Pysnooper.2*, both *Precision* and *Recall* were 100%. This indicates that the explanations precisely captured the input space affected by the patch, correctly predicting all patch-triggering inputs without misclassifying any non-triggering inputs. For other subjects this varies: For *Expression*, we achieved a *Precision* of 71% and a *Recall* of 93%. The high recall suggests that the explanation covered most of the patch-triggering input space, but the lower precision indicates that the explanation also included inputs outside the patch's input space, leading to false positives. Similarly, for *Cookiecutter.2* we recorded a *Precision* of 81% and a *Recall* of 93%. This shows a reasonably accurate approximation of the input space, with some over-approximation resulting in false positives.

Overall, the average *Precision* was 84%, and the average *Recall* was 94% across all subjects. This demonstrates that AVICENNAPATCH's explanations generally approximate the patch's input space accurately, effectively predicting whether inputs belong to the affected input space. However, in some cases, particularly with more complex subjects, the precision was lower due to false positives.

C. RQ2: Generating Patch-Triggered Inputs

To assess how effectively AVICENNAPATCH's explanations approximate the patch's input space by generating new patch-triggering inputs, we used the ISLA solver to generate new inputs and measured the *Precision* and *Recall*. The results are shown in Table I under the **Producer** columns.

For most subjects, the explanations proved highly effective in generating inputs that accurately represent the patch's input space: *Logarithm*, *Calculator*, *Middle.1*, *Middle.2*, *Pysnooper.1*, *Pysnooper.2*, and *Cookiecutter.3* all achieved 100% *Precision* and *Recall*. This indicates that the explanations precisely captured the input space affected by the patch, enabling the generation of inputs that consistently triggered the patch without including any non-triggering inputs. *Expression* exhibited a *Precision* of 100% but a lower *Recall* of 66%. The perfect precision means all generated inputs belonged to the patch's input space, but the lower recall suggests that the explanation did not cover all possible patch-triggering inputs, resulting in a partial approximation of the input space.

Overall, the high precision across all subjects indicates that the explanations effectively generate valid patch-triggering inputs without producing false positives. The variable recall reflects the challenge of fully covering the patch's input space, especially in programs with complex input formats or patch logic. These findings demonstrate that AVICENNAPATCH's explanations are generally effective in approximating the patch's input space when used as producers, generating new inputs that belong to the affected input space. In cases where recall was lower, such as with *Expression*, further refinement of the explanations might enhance coverage of the input space.

Summary of Results The evaluation results indicate that AVICENNAPATCH is generally effective in accurately approximating the input space affected by the patch. High *Precision* and *Recall* values across multiple subjects demonstrate the tool's capability to capture the conditions under which patches are activated. In the context of RQ1, for most subjects the explanations showed strong accuracy in predicting whether inputs belong to the patch-affected input space. Regarding RQ2, the explanations proved effective in generating new inputs that belong to the patch's input space. The generated inputs consistently triggered the patch as intended, confirming the explanations' utility in producing relevant test cases.

Overall, these findings suggest that AVICENNAPATCH performs well in approximating the input space of the patch, aiding developers in understanding and testing patches effectively.

D. Threats to Validity

1) Internal Validity: Our evaluation relies on the subjects and parameters provided by Tests4Py, which may influence the learning process of AVICENNAPATCH. If, for instance, inputs are not representative of the broader input space, the generated explanations might not generalize well. To mitigate this threat, we used the parameters supplied by Tests4Py, ensuring consistency with prior studies and benchmarks.

The iterative learning and refinement process in AVICENNAPATCH involves randomness, such as in input generation and learning. This could lead to variability in results across different runs. To address this, we repeated each experiment 10 times with different random seeds and reported average *Precision* and *Recall* values, thereby reducing the impact of random fluctuations.

2) External Validity: Our study evaluated AVICENNAPATCH on 12 bugs from 7 projects, which, while diverse, may not encompass the full spectrum of software systems and patches found in real-world applications. The subjects included both toy examples and real-world projects, but the sample size is still limited. Therefore, the results might not generalize to all types of software or patches, especially those with significantly different characteristics or complexities.

The performance of AVICENNAPATCH may be sensitive to specific configuration settings and parameters, such as the number of iterations in the learning loop or thresholds for precision and recall during constraint learning. We used default settings and justified our choices based on preliminary results, but different settings might yield different outcomes.

VI. LIMITATIONS

AVICENNAPATCH inherits several limitations from the AVICENNA framework. Chief among these is its reliance on an appropriate vocabulary to express the conditions under which a patch is triggered. The effectiveness of the generated explanations depends on the expressiveness and accuracy of this vocabulary.

Additionally, AVICENNAPATCH's use of grammars to generate new inputs imposes constraints on its applicability. Specifically, it cannot produce or explain patches targeting cases involving unexpected input types or formats outside the defined grammar. Since the grammar specifies the expected structure of input parameters, generating or reasoning about inputs that deviate from this structure is currently beyond AVICENNAPATCH's capabilities.

VII. CONCLUSION

In this paper, we introduced AVICENNAPATCH, a novel approach to understand and explain the conditions under which patches are activated. By leveraging a differential testing strategy, AVICENNAPATCH provides developers with clear insights into the specific input characteristics that trigger a patch, supporting the transparency, reliability, and predictability of automatic program repair. Through its differential outcome oracle, AVICENNAPATCH enables a nuanced understanding of patch behavior, distinguishing inputs that impact program outcomes from those that do not. This focus on patch activation helps developers evaluate the effectiveness and scope of patches, addressing a critical gap in APR by enhancing developers' ability to trust and validate automated fixes.

Several directions could further enhance AVICENNAPATCH's capabilities and impact:

Partial and Incomplete Patches. AVICENNAPATCH could be extended to identify partial or incomplete patches by analyzing the properties of failing inputs remaining in S_2. This would enable developers to pinpoint cases where

the patch does not fully address the issue, highlighting missing functionality or overly specific patch conditions for further refinement.

Integration with Symbolic Execution: Incorporating symbolic or concolic execution techniques could complement AVICENNAPATCH's input-based analysis. This integration would enhance its ability to explore input conditions for deeply nested or interdependent code paths, increasing the precision and comprehensiveness of input classifications.

ACKNOWLEDGMENT

This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG) under the projects Emperor (261444241) and IdeFix (496588242).

REFERENCES

[1] D. Steinhöfel and A. Zeller, "Input invariants," in *ESEC/SIGSOFT FSE*. ACM, 2022, pp. 583–594.
[2] OpenAI, "Chatgpt," 2024. [Online]. Available: https://chatgpt.com
[3] D. Beyer, L. Grunske, M. Kettl, M. L. Rosenfeld, and M. Raselimo, "P3: A dataset of partial program patches," in *MSR*. ACM, 2024, pp. 123–127.
[4] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *ICSE (1)*. ACM, 2010, pp. 55–64.
[5] P. D. Marinescu and C. Cadar, "KATCH: high-coverage testing of software patches," in *ESEC/SIGSOFT FSE*. ACM, 2013, pp. 235–245.
[6] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *DSN*. IEEE Computer Society, 2009, pp. 359–368.
[7] K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *SAS*, ser. Lecture Notes in Computer Science, vol. 6887. Springer, 2011, pp. 95–111.
[8] Z. Xu and G. Rothermel, "Directed test suite augmentation," in *APSEC*. IEEE Computer Society, 2009, pp. 406–413.
[9] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 3:1–3:42, 2014.
[10] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS*. ACM, 2017, pp. 2329–2344.
[11] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: directed grey-box fuzzing with provable path pruning," in *SP*. IEEE, 2022, pp. 36–50.
[12] C. Luo, W. Meng, and P. Li, "Selectfuzz: Efficient directed fuzzing with selective path exploration," in *SP*. IEEE, 2023, pp. 2693–2707.
[13] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "DAFL: directed grey-box fuzzing guided by data dependency," in *USENIX Security Symposium*. USENIX Association, 2023, pp. 4931–4948.
[14] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *CCS*. ACM, 2021, pp. 2169–2182.
[15] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *GECCO*. ACM, 2009, pp. 947–954.
[16] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*. IEEE, 2009, pp. 364–374.
[17] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM*, vol. 53, no. 5, pp. 109–116, 2010.
[18] M. Smytzek, M. Eberlein, K. Werk, L. Grunske, and A. Zeller, "Fixkit: A program repair collection for python," in *ASE*. ACM, 2024, pp. 2342–2345.
[19] H. Ruan, H. L. Nguyen, R. Shariffdeen, Y. Noller, and A. Roychoudhury, "Evolutionary testing for program repair," in *ICST*. IEEE, 2024, pp. 105–116.
[20] M. Martinez and M. Monperrus, "ASTOR: a program repair library for java (demo)," in *ISSTA*. ACM, 2016, pp. 441–444.
[21] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *ICST*. IEEE Computer Society, 2010, pp. 65–74.
[22] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA*. ACM, 2019, pp. 19–30.
[23] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*. IEEE Computer Society, 2013, pp. 802–811.
[24] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empir. Softw. Eng.*, vol. 25, no. 3, pp. 1980–2024, 2020.
[25] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *SANER*. IEEE Computer Society, 2018, pp. 118–129.
[26] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: scalable multiline program patch synthesis via symbolic analysis," in *ICSE*. ACM, 2016, pp. 691–701.
[27] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *ICSE*. IEEE Computer Society, 2013, pp. 772–781.
[28] X. D. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser, "JFIX: semantics-based repair of java programs via symbolic pathfinder," in *ISSTA*. ACM, 2017, pp. 376–379.
[29] X. Meng, X. Wang, H. Zhang, H. Sun, X. Liu, and C. Hu, "Template-based neural program repair," in *ICSE*. IEEE, 2023, pp. 1456–1468.
[30] Q. Zhu, Z. Sun, W. Zhang, Y. Xiong, and L. Zhang, "Tare: Type-aware neural program repair," in *ICSE*. IEEE, 2023, pp. 1443–1455.
[31] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *ASE*. ACM, 2022, pp. 92:1–92:13.
[32] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *ICSE*. ACM, 2022, pp. 1506–1518.
[33] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *ESEC/SIGSOFT FSE*. ACM, 2021, pp. 341–353.
[34] N. Jiang, T. Lutellier, and L. Tan, "CURE: code-aware neural machine translation for automatic program repair," in *ICSE*. IEEE, 2021, pp. 1161–1173.
[35] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *ISSTA*. ACM, 2020, pp. 101–114.
[36] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *ICSE*. IEEE, 2023, pp. 1482–1494.
[37] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *ESEC/SIGSOFT FSE*. ACM, 2022, pp. 959–971.
[38] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *ESEC/SIGSOFT FSE*. ACM, 2015, pp. 532–543.
[39] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 14:1–14:27, 2021.
[40] R. S. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *PLDI*. ACM, 2021, pp. 390–405.
[41] Y. Zhang, X. Gao, G. J. Duck, and A. Roychoudhury, "Program vulnerability repair via inductive inference," in *ISSTA*. ACM, 2022, pp. 691–702.
[42] M. Eberlein, M. Smytzek, D. Steinhöfel, L. Grunske, and A. Zeller, "Semantic debugging," in *ESEC/SIGSOFT FSE*. ACM, 2023, pp. 438–449.
[43] M. Smytzek, M. Eberlein, B. Serce, L. Grunske, and A. Zeller, "Tests4py: A benchmark for system testing," in *SIGSOFT FSE Companion*. ACM, 2024, pp. 557–561.
[44] R. Rachum, A. Hall, I. Yanokura *et al.*, "Pysnooper: Never use print for debugging again," 2019.
[45] A. R. Greenfeld, "Cookiecutter," 2022.
[46] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," in *The Fuzzing Book*. Saarland University, 2019. [Online]. Available: https://www.fuzzingbook.org/
[47] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 1556–1560.