Towards Interaction-Aware Validation Oracles for Multi-Fault Program Repair

Omar I. Al-Bataineh
Gran Sasso Science Institute
L'Aquila, Italy

***Abstract*—We explore the *oracle problem* in automated repair of multi-fault programs, focusing on validating partial patches produced during incremental repair phases. Given a multi-fault program P with faults $f_1, \ldots, f_n$ and a test suite T triggering these faults, the goal is to design validation oracles $\mathcal{O}^{(1)}, \ldots, \mathcal{O}^{(n)}$ for intermediate patches $pt_1, \ldots, pt_n$ that address individual faults.**

Validating partial patches raises two major issues. First, such patches are typically generated incrementally by APR tools and may address only a subset of the program's faults, leaving others unresolved. As a result, they often fail to produce correct outputs when evaluated in isolation. Second, validation must account for interactions with remaining faults to avoid rejecting valid fixes.

To address these issues, we outline a validation framework that blends the strengths of output-based, halting, and assertion-based oracles, some of which are informed by bug reports. The framework provides a modular approach to assess both partial and composite patches in programs with multiple defects by integrating formal reasoning and modeling the interactions of faults.

This conceptual framework lays the groundwork for more fault-aware and interaction-sensitive repair techniques and moves us toward a deeper, more practical understanding of how to assess patch correctness in complex, real-world multi-fault scenarios.

I. Introduction

Automated program repair (APR) has made significant advancements in recent years [1], [2], but most efforts still concentrate on addressing one fault at a time. In practice, however, real-world software often suffers from multiple faults that interact with one another in unexpected ways [3], [4]. These multi-fault situations present their own unique challenges—ones that today's program repair tools are not yet prepared to manage.

The oracle challenge: A key open problem in multi-fault APR is designing effective validation oracles [5], [6]. Traditional oracles assume complete correctness after a patch is applied. In contrast, for partial patches in multi-fault contexts, correctness must be assessed under uncertainty: patches may not fully restore expected outputs and must be validated in the presence of unresolved faults. Furthermore, partial patches may interfere with earlier fixes, necessitating multi-step validation. This highlights the need for modular and interaction-aware validation strategies.

Contribution: We present a formal validation model for multi-fault programs with interacting faults. The model is guided by an interaction relation $\mathcal{I}$, which captures how faults influence one another and enabling modular validation of partial patches. Each partial patch pt_i is checked using an assertion-based oracle $\mathcal{O}_i$, ensuring that fault f_i is addressed without disrupting prior fixes. For full-patch validation, we combine a halting oracle $\mathcal{O}_{\text{halt}}$ to ensure termination and an output-based oracle $\mathcal{O}_{\text{output}}$ to assess correctness. Our approach tackles the oracle problem by enabling modular reasoning and facilitating early detection of overfitting. To our knowledge, this is the first patch validation framework to integrate multiple oracle types for both subpatch and full-patch assessment in multi-fault settings.

II. Problem Statement

APR has advanced significantly in recent years, yet a core question remains unresolved: how can we determine with confidence whether a generated patch is *correct*? This longstanding challenge, known as the *oracle problem*, takes on new urgency as we move beyond the dominant simplifying assumption of single-fault programs and tackle the more realistic and far more common case of *multi-fault scenarios* in real-world software.

Most APR tools, whether traditional or machine learning (ML)-based, rely on test suites as their primary oracle. In principle, a patch that passes all tests in a validation suite is deemed valid. However, this assumption often breaks when faults interact, yielding misleading test outcomes. One fault may mask another, causing false positives, while others produce failures that surface only when jointly active. These scenarios highlight the limits of test-based validation and the need for fault-aware evaluation.

As a result of such misleading test outcomes, patches generated for multi-fault programs by standard APR approaches are prone to overfitting — addressing symptoms rather than root causes. Consequently, these approaches may introduce new regressions or leave existing faults unresolved. These fundamental challenges raise several pressing questions.

- *Interaction-awareness:* How should the nature of *fault interactions* influence our patch validation strategy?
- *Oracle source:* What *sources of information* (beyond test outcomes) can we rely on to act as trustworthy validation oracles in the context of multi-fault programs?
- *Scalability:* How can we ensure that validation techniques remain effective as multi-fault program complexity grows?

We argue that addressing the oracle problem for multi-fault programs requires more than incremental extensions: it demands a fundamental *redefinition* of patch correctness under fault interactions, laying the groundwork for validation-aware repair.

III. Patch Validation Model for Multi-Fault Repair

This section introduces a formal patch validation model for reasoning about patch validation in multi-fault programs. Our model characterizes the semantics of faults, patches, interactions, and oracles to support modular and scalable validation.

A. Multi-Fault Programs and Test Suites

Let P be a faulty program containing co-occurring faults — that is, faults that manifest within a single execution — denoted by $F = \{f_1, f_2, \ldots, f_n\}$, where each f_i represents a deviation from the intended behavior. Let $T = \{t_1, t_2, \ldots, t_m\}$ be a test

suite that exposes faults in F. Each test $t \in T$ returns a verdict in $\{\text{pass}, \text{fail}\}$, depending on the observable behavior of P in the presence of the faults and their interactions.

B. Fault Interaction Model

In multi-fault programs, faults can interact in non-additive ways that distort behavior and complicate both debugging and repair. We model these interactions as a labeled relation:

$$\mathcal{I} \subseteq F \times F \times \mathcal{R}, \quad \text{where } \mathcal{R} = \{\text{mask}, \text{synergy}, \text{indep}, \text{cascade}\}$$

Here, F denotes the set of faults, and each triple (f_i, f_j, r) indicates that fault f_i *influences* fault f_j via a directed interaction of type $r \in \mathcal{R}$. We visualize this as a labeled edge $f_i \xrightarrow{r} f_j$, capturing the flow of influence. For each interaction type r, we define a corresponding subrelation:

$$\mathcal{I}_r = \{(f_i, f_j) \mid (f_i, f_j, r) \in \mathcal{I}\}$$

We consider the following canonical interaction types:

- *Masking:* Fault f_i suppresses or obscures the manifestation of fault f_j, rendering it unobservable when both are active.
- *Synergy:* Faults f_i and f_j jointly produce a failure not triggered by either in isolation.
- *Independence:* Faults f_i and f_j affect disjoint parts of the system and exhibit no observable interaction.
- *Cascading:* Fault f_i enables or amplifies the manifestation of fault f_j, increasing its visibility or severity.

These interactions can obscure failure signals, mislead localization efforts, and compromise patch validation. Modeling the interaction relation $\mathcal{I}$ and its subrelations is thus essential for reasoning about correctness and behavior in multi-fault scenarios.

C. Patches and Composition Semantics

Let $\mathsf{patch} = \{pt_1, pt_2, \ldots, pt_n\}$ be a set of candidate *subpatches*, where each pt_i is synthesized to target fault f_i. Subpatches can be composed using a binary operator $\oplus$ that merges two subpatches into a single patch[1] :

$$\mathsf{patch}_{\text{composite}} = pt_1 \oplus pt_2 \oplus \cdots \oplus pt_n.$$

The semantics of $\oplus$ depend on patch representations (e.g., AST rewrites, instruction-level edits) and govern the preservation or interference of earlier fixes. Fault interactions influence the success of both individual and composed patches.

D. Test and Validation Oracles

To validate patches, we often rely on an available test oracle

$$\mathcal{O}_T : T \times P \to \{\text{pass}, \text{fail}\}$$

which observes the behavior of program P on test input $t \in T$. While widely used, such oracles are not sufficient in multi-fault contexts, where passing tests may mask unresolved faults. We therefore define a higher-level *patch validation oracle*:

$$\mathcal{O}_V : T \times \mathsf{patch} \to \{\text{valid}, \text{invalid}\}$$

which assesses whether applying a patch (partial or full) satisfies the intended specification, potentially inferred from test behavior, formal contracts, or semantic models.

[1]Even when subpatches are generated incrementally, e.g., following an interaction-aware order, a final composition step is needed to integrate fixes for independent faults, some of which may be synthesized in parallel, and to check for emergent interactions not observable during individual validation.

E. Validation in Progressive Repair

Multi-fault repair is often conducted incrementally. A repair system R proceeds in phases, generating and validating subpatches pt_i, each intended to address a specific fault $f_i \in F$. However, validating a subpatch in the presence of other unaddressed faults is nontrivial, due to potential fault interactions and the incomplete correctness of intermediate program states.

To ensure sound validation during progressive repair, we define a formal notion of *valid partial patch* grounded in fault-aware semantics. We assume that the repair process respects a partial order $\prec_{\mathcal{I}}$ over F, induced by $\mathcal{I}$, where $f_j \prec_{\mathcal{I}} f_i$ implies that f_j should be addressed before f_i to avoid semantic interference.

Definition 1: **(Valid partial patch under interaction-aware ordering).** Let $\prec_{\mathcal{I}}$ be a partial order over faults $F = \{f_1, \ldots, f_n\}$ induced by the relation $\mathcal{I}$. A subpatch pt_i is deemed valid for fixing fault f_i in a multi-fault program P if:

1) It eliminates the faulty behavior directly attributed to f_i,
2) It does not adversely interfere with any previously validated subpatch pt_j for faults f_j such that $f_j \prec_{\mathcal{I}} f_i$.
3) It preserves the program state in a way that does not hinder the detection or repair of any fault f_k such that $f_i \prec_{\mathcal{I}} f_k$.

This definition ensures that each repair step contributes to the overall goal without regressing earlier fixes or compromising future ones. The interaction-aware ordering $\prec_{\mathcal{I}}$ guarantees that fault interdependencies, such as masking, synergy, and cascading, are respected in both validation and repair sequencing. Notably, a valid partial patch does not need to fix all failing tests, as some may stem from remaining faults, but it must avoid behaviors that obscure or disrupt the validation of later subpatches.

F. The Oracle Tuple Problem

A core challenge in multi-fault repair is designing validation oracles for partial patches that operate in the presence of complex fault interactions. Unlike single-fault repair, multi-fault scenarios demand a modular yet flexible validation approach.

*Definition 2 (**Oracle tuple for multi-fault validation**):* Let $\mathcal{O}_V = \langle \mathcal{O}_V^{(1)}, \ldots, \mathcal{O}_V^{(m)} \rangle$ be a tuple of validation oracles applied to subpatches $pt_1, \ldots, pt_n$, with $m \geq 1$. Each oracle $\mathcal{O}_V^{(i)}$ may target one or more faults and associated subpatches, possibly overlapping with others. We refer to $\mathcal{O}_V$ as an *oracle tuple* for program P under interaction model $\mathcal{I}$.

This formulation acknowledges that oracle-to-subpatch mappings are not strictly one-to-one. Some faults may require multiple oracles due to complexity or limited observability, while others may share a common oracle. The interaction model $\mathcal{I}$ is essential for designing and interpreting these oracles.

G. The Patch Validation Problem

Building on the formal validation model, we now define the *patch validation* problem for multi-fault programs:

Given a faulty program P with a set of faults $F = \{f_1, \ldots, f_n\}$, a test suite T, a set of candidate subpatches $\mathsf{patch} = \{pt_1, \ldots, pt_n\}$, a fault interaction model $\mathcal{I} \subseteq F \times F \times \mathcal{R}$, and a corresponding partial order $\prec_{\mathcal{I}}$ over F, the goal is to design a validation strategy V for P that can:

1) Assess whether each subpatch pt_i constitutes a valid intermediate fix when executed in the presence of other unresolved faults and their interactions;

2) Determine whether the full composition of subpatches, denoted $\texttt{patch}_{\text{composite}}$, yields a correct repair of P.

This formulation captures the central challenge of validating patches in multi-fault settings: the validity of a fix may depend on the behavioral context of other unresolved faults.

H. Research Challenges

The presented formulation raises several open challenges:

- *Partial validation:* How to validate the subpatch pt_i in isolation or in context, given interactions in $\mathcal{I}$?
- *Oracle construction:* How can we go beyond test-based oracles to build lightweight yet fault-aware validators?
- *Validation efficiency:* How many oracles need to be evaluated when applying a new subpatch $pt_i(P)$, and which ones? Can we leverage the interaction model $\mathcal{I}$ to avoid redundant validations and reduce computational overhead?
- *Scalability:* How to scale the validation process for large programs with many faults and vast patch spaces?

Efficiency and scalability in patch validation hinge on informed choices about *which* oracles to invoke and *when*. The number and selection of oracles for a subpatch $pt_i(P)$ should be guided by the fault interaction model $\mathcal{I}$, not chosen arbitrarily. For instance, if pt_i targets fault f_i and $\mathcal{I}$ indicates interaction with another fault f_j, then oracles for both should be used to catch potential side effects. If f_i is independent, validating it in isolation may suffice. This demonstrates how incorporating $\mathcal{I}$ can reduce redundant testing and improve scalability in multi-fault repair.

IV. Instantiating the Patch Validation Model

To operationalize our patch validation model, we introduce a workflow that integrates fault estimation, interaction analysis, and modular validation. As shown in Fig. 1, the process begins with a faulty program and its test suite, followed by estimating the fault set F, analyzing interactions $\mathcal{I}$, and deriving a partial order $\prec_{\mathcal{I}}$. These guide the synthesis of sub-oracles that drive iterative subpatch generation. The model supports refinement, allowing updates as new interactions are discovered. The final composite patch is validated using both output-based and halting oracles. Each phase is detailed in the remainder of this section.

A. Identifying the Fault Set F

Traditional fault localization techniques are typically designed for single-fault scenarios, where a single root cause is assumed to underlie observed failures. In multi-fault programs, however, multiple faults can obscure, amplify, or distort fault signals, making accurate localization significantly more difficult.

To address this, we propose a hybrid approach that combines AI-based models with multi-fault localization (MFL) techniques [7], [8], [9], [10]. AI-guided components, such as learning-based or statistical models trained on historical defect data, can rank program elements according to fault-proneness patterns. These initial candidates are then refined using dynamic MFL analyses like spectrum-based localization or dynamic slicing, which correlate runtime behavior with test failures.

This combined strategy yields a more accurate approximation of the fault set $F = \{f_1, f_2, ..., f_n\}$, which underpins subsequent modular patching and validation. Existing MFL tools, such as CFaults [11] and FLITSR [8], can be integrated to further enhance candidate identification in complex scenarios. Emerging

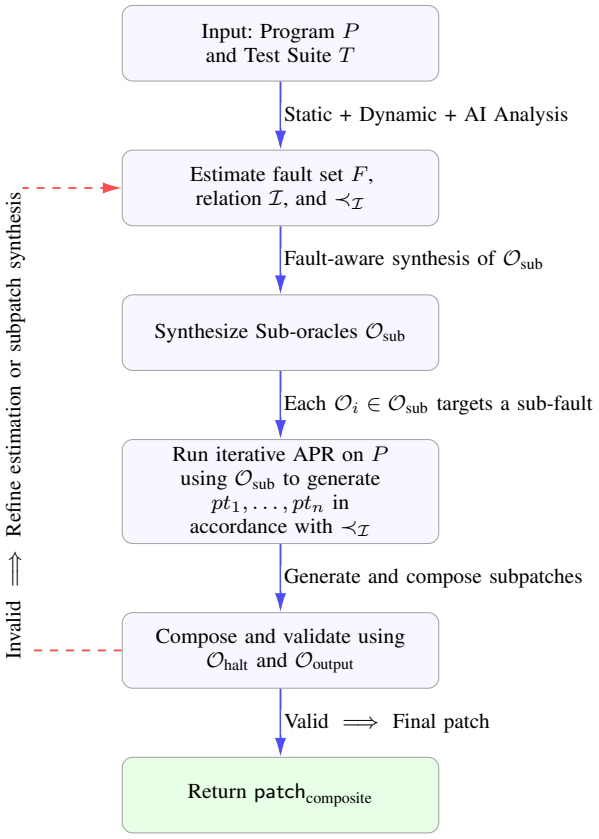

Fig. 1. Modular patch synthesis and validation workflow. Subpatches are generated iteratively based on $\prec_{\mathcal{I}}$ and validated in composition. Dashed arrow denotes refinement upon validation failure.

AI assistants, such as ChatGPT and Gemini, have also shown early promise in locating faults per session [12], [13], offering complementary support to programmatic MFL techniques.

B. Estimating the Interaction Relation $\mathcal{I}$

To make our patch validation model practical, we must estimate the interaction relation $\mathcal{I}$, which describes how faults in a multi-fault program influence one another. This relation governs how subpatches should be validated and composed. We outline several promising approaches to estimate the relation $\mathcal{I}$.

Dynamic estimation via subpatch experimentation: A lightweight and practical method to approximate $\mathcal{I}$ is through dynamic test executions guided by subpatch manipulation. By selectively applying or disabling subpatches and observing test outcomes, we can infer how faults interact. For instance:

- *Masking:* Applying patch pt_i alters test failure behavior, revealing that the fixed fault concealed the effects of another fault, which may have been hidden by crashes or hangs.
- *Synergy:* Only the combination of patches pt_i and pt_j causes a failing test to pass, though neither works alone.
- *Independence:* Patches pt_i and pt_j can be applied in isolation without mutual influence on test outcomes.
- *Cascading:* Applying patch pt_i unexpectedly causes a test to fail by exposing faulty behavior elsewhere, indicating a downstream dependency between faults.

Static and semantic augmentation: Static code analysis (e.g., control/data flow, aliasing, and dependence graphs) can complement dynamic observations by identifying syntactic or

semantic proximity between faults. For example, faults in the same control structure or sharing variables may have a higher likelihood of interaction. Semantic fault types, such as memory violations or hangs, can also be used to predict masking behavior (e.g., a crash will likely suppress output anomalies).

Graph-based interaction modeling: We can model estimated interactions as a labeled fault graph, where nodes represent faults and edges indicate observed or inferred interactions. These edges can be labeled with interaction types and optionally weighted by confidence or severity (e.g., degree of masking).

C. Fault Types and Validator Design

The design of sub-oracles in a multi-fault context must account not only for the behavior of individual faults but also for their interactions, as formalized by the relation $\mathcal{I}$. We classify faults into four categories, each requiring a tailored oracle construction:

- **Disruptive faults** (e.g., crashes, hangs) cause coarse-grained, non-functional failures. These require oracles that monitor liveness, exceptions, or runtime signals. For each disruptive fault f_i, we create an oracle $\mathcal{O}_i$ using runtime monitors and failure signatures from logs or bug reports. *Oracle count: one per fault. Validation: single-stage.*
- **Independent faults** exhibit no observable interaction with others (i.e., they do not participate in any tuple in $\mathcal{I}$). Their oracles rely on output-based comparisons using test cases. *Oracle count: one per fault. Validation: parallelizable.*
- **Synergistic faults** show non-decomposable behavior: their impact arises when multiple faults co-occur. If $(f_i, f_j) \in \mathcal{I}_{\text{synergy}}$, we construct a joint oracle $\mathcal{O}_{i,j}$ based on failure-inducing tests specific to their combination. *Oracle count: one per synergistic fault group. Validation: joint.*
- **Dependent faults (masked or cascading)** involve suppression or amplification—i.e., $(f_i, f_j) \in \mathcal{I}_{\text{mask}} \cup \mathcal{I}_{\text{cascade}}$. Validation is staged: dominant faults handled first, with secondary oracles activated as effects emerge. For dependency faults, we define an oracle dependency graph $G = (F, E)$, where $E = \mathcal{I}_{\text{mask}} \cup \mathcal{I}_{\text{cascade}}$ guides validation order. *Oracle count: layered. Validation: sequential and interaction-aware.*

This structured oracle design aligns validation strategies with the behavioral complexity of real-world multi-fault programs. By tailoring sub-oracles to fault types and their interactions, we improve fault isolation, reduce false positives, and enable more robust patch validation. Ultimately, this modular framework supports effective integration into APR pipelines.

D. Validating the Full Patch: Composite Oracles for Soundness

While sub-oracles can detect early signs of invalid partial repairs, they are insufficient to guarantee the correctness of a complete patch that integrates multiple subpatches. To ensure the reliability of the final full patch, we advocate a composite validation strategy employing two complementary oracles:

- *Output-based oracle $\mathcal{O}_{output}$:* This oracle compares the observable outputs of the patched program against expected results. It captures functional correctness and is suitable for detecting incorrect behavior in non-disruptive faults.
- *Halting oracle $\mathcal{O}_{halt}$:* This oracle checks that the final program terminates under all inputs, detecting crash- or hang-inducing behaviors. $\mathcal{O}_{\text{halt}}$ can be checked using termination provers (e.g., AProVE [14], T2 [15]), which may return *terminating* (`TR`), *non-terminating* (`NT`), or *unknown* (`UN`).

Validating the final full patch with both oracles strengthens the overall assessment by covering complementary aspects of correctness: $\mathcal{O}_{\text{halt}}$ ensures liveness and safety, while $\mathcal{O}_{\text{output}}$ checks for accuracy in the program's observable behavior.

Refinement upon validation failure: In multi-fault repair, it is possible that all subpatches pass their respective sub-oracles, yet the final full patch fails global validation. Such contradictions often indicate hidden fault interactions, flawed assumptions in the interaction model, or overfitting in subpatches. A refinement loop should be triggered in these cases, which may involve revisiting the fault interaction assumptions, adjusting sub-oracle scopes, or resynthesizing subpatches with updated constraints. Supporting such iterative refinement is essential to ensure correctness in the presence of complex or synergistic fault behaviors.

Bug reports and oracle design: In practice, bug reports often include stack traces, reproduction steps, and descriptions of faulty behavior [16], [17], [18]. These can be leveraged to derive lightweight assertion-based oracles, especially for disruptive faults. Such oracles are valuable not only for subpatch validation but also as auxiliary signals during full-patch evaluation.

Table I summarizes how validation strategies differ by fault type, interaction pattern, and oracle source. It contrasts local and global validation goals, identifies when refinement is needed, and maps each strategy to practical oracle derivation methods.

V. Demonstrating Example

To demonstrate the benefits of our validation model, we present a simplified yet realistic example based on sensor logic. It includes three faults capturing common interaction patterns: a disruptive crash, a semantic miscalculation, and an over-aggressive smoothing. These faults exhibit masking and synergy, two interaction types that challenge traditional patch validation.

```
1  float process_reading(int sensor_val) {
2      int offset = sensor_val % 10 - 5;
3      // f1: Risky zero divisor
4      int normalized = sensor_val / offset;
5      // f1: Div-by-zero if offset == 0
6      float temp = normalized * 1.5 - 32;
7      // f2: Wrong conversion formula
8      if (temp < 50.0)
9          temp = round(temp / 10.0) * 10.0;
10     // f3: Excessive rounding
11     return temp;
12 }
```

Listing 1. Three interacting faults in sensor processing

A. Fault Interaction Matrix

To illustrate the fault interactions in our example (Listing 1), we present the *fault interaction matrix* IM_P, a concise tabular view derived directly from the interaction relation $\mathcal{I}$. Each entry at row f_i, column f_j corresponds to a labeled interaction $(f_i, f_j, r) \in \mathcal{I}$, capturing how fault f_i influences fault f_j. This matrix encodes directed fault interactions and may be asymmetric, reflecting potentially different effects in opposite directions.

$IM_P =$

	f_1	f_2	f_3
f_1	–	mask	mask
f_2	indep	–	synergy
f_3	indep	synergy	–

TABLE I
VALIDATION STRATEGIES ACROSS FAULT TYPES, PATCH LEVELS, AND ORACLE SOURCES.

Fault Type	Subpatch Strategy	Full-Patch Strategy	Validation Purpose	Oracle Source(s)
Disruptive	Halting checks (timeouts, crash monitors)	Combined $\mathcal{O}_{\text{halt}} + \mathcal{O}_{\text{output}}$	Ensure termination and functional correctness	Termination provers, test outputs, crash reports
Independent	One oracle per fault (assertion or output-based)	$\mathcal{O}_{\text{output}}$	Validate fault-local fixes and isolate behavior	User-provided test suite, inferred postconditions
Synergistic	Compound sub-oracle over joint fault group	$\mathcal{O}_{\text{output}} + \mathcal{O}_{\text{halt}}$	Detect combined behavior or non-decomposable fixes	Test outcomes, user assertions, joint failure analysis
Masked	Oracle refinement guided by masking; dynamic toggling	$\mathcal{O}_{\text{output}}$	Reveal suppressed behavior due to other faults	Test logs, bug traces, patch experiments
Cascading	Oracle refinement staged with patch order	Ordered composite validation using both oracles	Reveal dependent or emergent faults post-fix	Patch ordering traces, dynamic monitors
Validation Failure	—	Refinement loop on $\mathcal{O}_{\text{halt}}/\mathcal{O}_{\text{output}}$ failure	Diagnose fault interaction misfit or subpatch overfitting	Inferred from failed validation outcomes

The diagonal entries are marked as –, indicating that faults are not considered to interact with themselves.

Fault `f1` is disruptive and exhibits masking behavior: it halts execution prematurely, suppressing the manifestations of `f2` and `f3`. In contrast, `f2` and `f3` are synergistic: each causes only minor deviations individually, but together they lead to significant output errors that exceed acceptable thresholds.

B. Validation Analysis

We assume the availability of: (i) a failing test suite T, (ii) a bug report with runtime error information, (iii) a termination prover TP, and (iv) tools for fault localization and assertion inference (e.g., symbolic analyzers or AI-based assistants). Given this setup, we walk through the process of fault interaction discovery and oracle-guided patch validation.

1) **Detecting the disruptive fault `f1`**: Fault `f1` triggers a division-by-zero crash when `offset == 0`, which occurs for inputs like 85, 95, or 105. This crash is easily detectable via dynamic analysis or lightweight static checks. Even general-purpose tools (e.g., ChatGPT) can localize the fault based on runtime output like "Floating point exception at line 3." Since the crash halts execution, it also masks faults `f2` and `f3`. Recognizing this masking behavior is essential: validating a fix for `f1` must not prematurely reject correct patches just because `f2` or `f3` remain unpatched.
2) **Unmasking faults `f2` and `f3`**: After `f1` is fixed, program normal execution resumes, and the remaining faults manifest. Fault `f2` applies an incorrect conversion from sensor units to temperature, while fault `f3` rounds aggressively, reducing precision. Both faults produce observable deviations only when the program runs to completion.
3) **Suboracle validation**: Based on the discovered fault set $F = \{f_1, f_2, f_3\}$ and their interaction matrix IM_P, we construct suboracles tailored to each fault's characteristics:
   - *For `f1`*: we use two complementary oracles:
     - $\mathcal{O}_{\text{halt}}$: A halting oracle ensures the program no longer crashes. It can be validated using termination provers like AProVE or Ultimate Automizer.
     - $\mathcal{O}_{\text{assert}}$: An assertion oracle is derived from runtime symptoms and bug reports. For instance, the report may include "Floating point exception at line 3", suggesting an assertion like `assert(offset != 0);`. We can use tools like CPAChecker [19] or Frama-C [20] to check the validity of the assertion.
   - *For `f2` and `f3`*: since they produce incorrect but terminating output, we validate patches using:
     - $\mathcal{O}_{\text{output}}$: This is a traditional output oracle derived from the user-provided test suite T, optionally enhanced by test amplification or fuzzing techniques to better capture semantic deviations.

   Each suboracle aligns with the fault's role in the interaction matrix: crash faults require halting/assertion oracles; synergistic semantic faults demand nuanced output oracles. Crucially, these suboracles allow us to validate partial repairs without false rejection due to masked behavior.
4) **Final patch validation**: Once all faults have been repaired and their respective subpatches validated, we perform a final, integrated validation using the combination $\mathcal{O}_{\text{halt}} + \mathcal{O}_{\text{output}}$. This ensures the patched program both terminates correctly and produces expected results across the full test suite.

This example illustrates how our oracle-aware, interaction-informed validation model enables practical, modular assessment of multi-fault repairs. By tailoring suboracle construction to fault interaction types, we reduce both underfitting and overfitting risks, improving validation precision and boosting developer confidence in automated patch correctness.

VI. CONCLUSION

Automated repair of multi-fault programs with co-occurring bugs poses a growing challenge in software engineering. This paper tackles a key yet underexplored issue: the oracle problem in multi-fault repair, focusing on constructing oracles to validate candidate patches addressing several faults simultaneously.

We propose an oracle-aware validation model that integrates three oracle types: output-based, halting, and assertion-based. The output and halting oracles jointly validate the final composite patch for correctness and termination, while assertion-based oracles guide validation of subpatches targeting individual faults.

Current APR validation strategies are fundamentally unreliable in multi-fault scenarios: dependence on test suites weakens and misleads patch validation due to fault interactions. Our model addresses this issue by examining the interactions among faults, fault characteristics, and validation strategies, thereby reducing overfitting, fault masking, and emergent behaviors.

We hope this work inspires deeper exploration of oracle design in repair workflows and promotes the development of more robust repair tools for real-world multi-fault programs.

REFERENCES

[1] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, Nov. 2019.

[2] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[3] O. I. Al-Bataineh, "Automated repair of multi-fault programs: Obstacles, approaches, and prospects," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 2215–2219.

[4] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *37th IEEE/ACM International Conference on Software Engineering, ICSE*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 913–923.

[5] C. Geethal, M. Böhme, and V.-T. Pham, "Human-in-the-loop automatic program repair," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4526–4549, 2023.

[6] O. I. Al-Bataineh, "Towards developing effective oracles to reduce patch overfitting in automated program repair," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2025.

[7] S. Xu, Y. Gao, X. Cai, Z. Wang, and H. Ji, "Effective multi-fault localization based on fault-relevant statistics," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2021, pp. 998–1003.

[8] D. Callaghan and B. Fischer, "Improving spectrum-based localization of multiple faults by iterative test suite reduction," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 1445–1457.

[9] D. Ghosh and J. Singh, "Spectrum-based multi-fault localization using chaotic genetic algorithm," *Inf. Softw. Technol.*, vol. 133, p. 106512, 2021.

[10] S. Lamraoui and S. Nakajima, "A formula-based approach for automatic fault localization of multi-fault programs," *J. Inf. Process.*, vol. 24, no. 1, pp. 88–98, 2016.

[11] P. Orvalho, M. Janota, and V. M. Manquinho, "cfaults: Model-based diagnosis for fault localization in C with multiple test cases," in *Formal Methods - 26th International Symposium, FM*, ser. Lecture Notes in Computer Science, vol. 14933. Springer, 2024, pp. 463–481.

[12] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–47, 2024.

[13] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1469–1481. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00128

[14] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann, "Proving Termination of Programs Automatically with AProVE," in *International Joint Conference on Automated Reasoning (IJCAR)*, Cham, 2014, pp. 184–191.

[15] H.-Y. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter, "Synthesising Interprocedural Bit-Precise Termination Proofs (T)," in *International Conference on Automated Software Engineering (ASE)*, 2015, pp. 53–64.

[16] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *28th International Conference on Software Engineering (ICSE)*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 361–370.

[17] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. L. Traon, "ifixr: bug report driven program repair," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*, 2019, pp. 314–325.

[18] M. Motwani and Y. Brun, "Better automatic program repair by using bug reports and tests together," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00109

[19] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Computer Aided Verification*, 2011, pp. 184–190.

[20] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c - A software analysis perspective," in *Software Engineering and Formal Methods - 10th International Conference, SEFM*, vol. 7504, 2012, pp. 233–247.