Debugging the Undebuggable: Why Multi-Fault Programs Break Debugging and Repair Tools

Omar I. Al-Bataineh
Gran Sasso Science Institute, L'Aquila, Italy
omar.albataineh@gssi.it

***Abstract*—Multi-fault programs, which contain more than one bug simultaneously, are notoriously difficult to debug and repair. This is largely because faults can interact in subtle ways: one might hide the effects of another, or even cause new failures to appear when combined. In this paper, we investigate why multi-fault programs remain so challenging for today's debugging and repair tools. We introduce a formal model that captures the different ways faults can interact, including masking, synergy, and cascading. Building on this model, we propose a novel framework for reasoning about faults, not in isolation, but as part of a network of influences. This perspective opens the door for future tools that can better understand, diagnose, and repair programs with multiple faults.**

I. Introduction

Program debugging is a fundamental task in software engineering, aiming to automatically locate and repair program faults. While much prior research has focused on *single-fault* programs, where only one fault is assumed to be present, real-world software systems often exhibit *multiple co-occurring faults*. These faults can interact in complex and often unpredictable ways, posing significant challenges to state-of-the-art debugging and automated program repair techniques [1], [2], [3].

This paper tackles a critical challenge in program testing, debugging, and repair: *why multi-fault programs remain difficult to handle*. To explore this, we introduce a formal model that illustrates how faults interact, including masking each other, creating new failure modes, or triggering unexpected behavior. We identify four primary interaction types: *independence*, *masking*, *synergy*, and *cascading*, which demonstrate how interactions undermine the assumptions of traditional fault localization and repair tools. Our model clarifies, for example, why fixing one fault can inadvertently expose or intensify others, complicating the repair process. Although emerging AI-powered repair tools show promise in managing these interactions, they lack explicit reasoning about fault interaction. Our work establishes the groundwork for "fault-aware" debugging and repair, enabling future tools, both conventional and AI-based, to better handle the complexities of multi-fault programs.

Impact on SE activities: A clear understanding of how faults interact can substantially enhance the way we test, debug, and repair software. Instead of viewing faults as isolated glitches, this work emphasizes the need for analysis techniques that can detect and reason about interactions among faults. This requires methods that consider how code is structured, how data flows through it, and how it behaves at runtime. With such insight, we may develop tools that not only detect individual defects but also anticipate their interactions, allowing developers to discover hidden effects and provide more accurate fixes.

II. Formal Model

In this section, we formalize a set of interaction types that help explain why certain faults are difficult to localize or repair. Our main goal is to highlight how these interactions impact observability and interfere with automated reasoning about faults. To motivate this formalization, we begin by examining why existing testing techniques struggle in multi-fault settings.

A. Limitations of Traditional Testing Techniques

Traditional testing techniques encounter significant challenges when applied to multi-fault programs. First, they often fail to accurately differentiate between multiple faults occurring simultaneously, which makes isolating individual issues more difficult. Second, they are inadequate at managing masking faults, where one fault conceals or prevents the activation of another, resulting in incomplete diagnostics and misplaced confidence in repairs. Third, estimating the total number of faults becomes unreliable when fault interactions alter observable behaviors, complicating efforts to evaluate test completeness or fault coverage. Finally, the computational complexity involved in analyzing multi-fault programs escalates quickly, as the number of potential fault interactions increases exponentially, making it hard to scale existing test generation or analysis techniques.

These limitations reveal the need for a deeper, formal understanding of how faults interact and how such interactions fundamentally undermine current testing, localization, and repair.

B. Definitions and Setup

Let P be a program with a set of faults $F = \{f_1, f_2, \ldots, f_n\}$, and let T be a test suite that exposes each fault $f_i \in F$. To reason about how faults behave and interact under different settings, we define the following functions:

- *Execution:* $\mathsf{exec}(f_i, P[C], T) \in \{\text{false}, \text{true}\}$ denotes whether fault f_i is executed under any test in T, when program P is instrumented with the fault set $C \subseteq \mathcal{F}$, meaning the faults in C are actively present in the program during testing.
- *Visibility:* $v(f_i, P[C], T) \in \{\text{false}, \text{true}\}$ indicates whether fault f_i causes an observable failure in at least one test from T when evaluated in the context of fault set C.
- *Behavior:* $beh(f_i, C) \in \{\mathsf{C}, \mathsf{H}, \mathsf{E}, \mathsf{L}, \mathsf{N}\}$ represents the failure mode caused by fault f_i when active with fault set C. The values are: C for crash, H for hang, E for uncaught exception, L for logic error, and N for no observable failure. We define the set of disruptive behaviors as $\mathsf{D} = \{\mathsf{C}, \mathsf{H}, \mathsf{E}, \mathsf{L}\}$.

These functions capture fault execution, observability, and behavior, allowing us to formalize when two faults *interact*.

*Definition 1 (**Interacting Faults**):* Let P be a program containing two distinct faults, f_i and f_j. We say that f_i and

f_j are *interacting faults* if there exists at least one program execution trace in which the presence or repair of one fault affects the manifestation, observability, or repairability of the other. Formally, this interaction means:

- The *behavior* of P with fault f_i present depends on whether f_j is also present or repaired, or vice versa;
- The *observability* of one fault's effects in test executions depends on the presence or repair status of the other fault;
- The *ability to repair* one fault (i.e., generate a correct patch) is affected by the presence or repair of the other fault.

Such dependencies indicate that the faults cannot be treated independently during fault localization or program repair.

These formal elements allow us to characterize fault behavior, both individually and in combination. Faults are not isolated entities; they may suppress, expose, or activate one another. To illustrate, consider the following example:

Illustrative Example.: Suppose f_1 is a null pointer dereference occurring early in the execution path, and f_2 is a missing input check later in the same path. In isolation, f_2 may cause a logic error (e.g., invalid computation), and f_1 causes an immediate crash. However, when both faults are active, f_1 causes the program to crash before f_2's logic error can manifest. This is an instance of *masking*, where a disruptive failure caused by one fault hides the effects of another. Such scenarios are common in practice, yet current debugging techniques often ignore them.

C. Fault Interaction Types

We define an interaction relation $\mathcal{I}(f_i, f_j, P, T)$ that classifies the relation between faults f_i and f_j in P under test suite T. The following four fault interaction types are considered:

- **Independence.** Faults f_i and f_j are *independent* if their effects are isolated, each fault is both visible and functionally stable, whether it appears alone or together with the other:

$$\begin{gathered}\mathcal{I}(f_i, f_j, P, T) = \mathsf{indep} \iff \\ v(f_i, P[\{f_i\}], T) \wedge v(f_j, P[\{f_j\}], T) \\ \wedge\, v(f_i, P[\{f_i, f_j\}], T) \wedge v(f_j, P[\{f_i, f_j\}], T) \\ \wedge\, beh(f_i, \{f_i\}) = beh(f_i, \{f_i, f_j\}) \\ \wedge\, beh(f_j, \{f_j\}) = beh(f_j, \{f_i, f_j\})\end{gathered}$$

  Fault independence reflects the ideal scenario for debugging and repair tools: each fault behaves as if the others were not present. That is, the presence of one fault neither masks nor alters the effect of another. This separation enables faults to be diagnosed and addressed individually, without accounting for complex interactions. Our formal definition enforces two key properties: *visibility*, meaning that each fault still leads to a detectable failure; and *behavioral stability*, meaning that the nature of that failure remains consistent. These conditions rule out cases where faults subtly interfere, even when both are triggered concurrently.

- **Masking.** Fault f_i *masks* fault f_j if it causes a severe failure that suppresses f_j's observability when both are present:

$$\begin{gathered}\mathcal{I}(f_i, f_j, P, T) = \mathsf{mask} \iff \\ v(f_i, P[\{f_i\}], T) \wedge beh(f_i, \{f_i\}) \in \mathsf{D} \\ \Rightarrow \neg v(f_j, P[\{f_i, f_j\}], T) \quad \text{for } j \neq i\end{gathered}$$

  Masking occurs when one fault dominates the execution with a disruptive failure, effectively hiding the presence of another co-occurring fault. While f_j might be visible and detectable when evaluated in isolation, its manifestations become unobservable in the presence of f_i. This interaction can mislead debugging and automated repair by producing an incomplete picture of the fault landscape. Properly identifying masking helps avoid underdiagnosis and ensures more comprehensive and fault-aware repair strategies.

- **Synergy.** Faults f_i and f_j exhibit *synergy* if their behaviors combine additively when they occur together, leading to a failure that neither fault causes individually. Specifically, the faults must both be present in the same program execution, and their combined effect results in a failure:

$$\begin{gathered}\mathcal{I}(f_i, f_j, P, T) = \mathsf{synergy} \iff \\ v(f_i, P[\{f_i\}], T) \wedge v(f_j, P[\{f_j\}], T) \\ \wedge\, beh(f_i, \{f_i\}) \notin \mathsf{D} \wedge beh(f_j, \{f_j\}) \notin \mathsf{D} \\ \wedge\, v(f_i, P[\{f_i, f_j\}], T) \wedge v(f_j, P[\{f_i, f_j\}], T) \\ \wedge\, (beh(f_i, \{f_i, f_j\}) \in \mathsf{D} \vee beh(f_j, \{f_i, f_j\}) \in \mathsf{D})\end{gathered}$$

  Fault synergy occurs when the combined effect of f_i and f_j causes a failure, even though each fault is harmless or non-disruptive when considered independently. This additive behavior complicates fault localization and APR systems, as one fault may obscure or enhance the impact of the other. Fault synergy is a significant challenge in multi-fault programs, where seemingly benign or isolated faults can lead to substantial failures when they occur together.

- **Cascading.** Fault f_i *cascades* to f_j if it alters the control or data flow such that f_j, previously dormant, becomes executed and observable as a result of f_i's presence:

$$\begin{gathered}\mathcal{I}(f_i, f_j, P, T) = \mathsf{cascade} \iff \\ v(f_j, P[\{f_i, f_j\}], T) \wedge \neg v(f_j, P[\{f_j\}], T) \\ \wedge\, \mathsf{exec}(f_j, P[\{f_i, f_j\}], T) \wedge \neg \mathsf{exec}(f_j, P[\{f_j\}], T)\end{gathered}$$

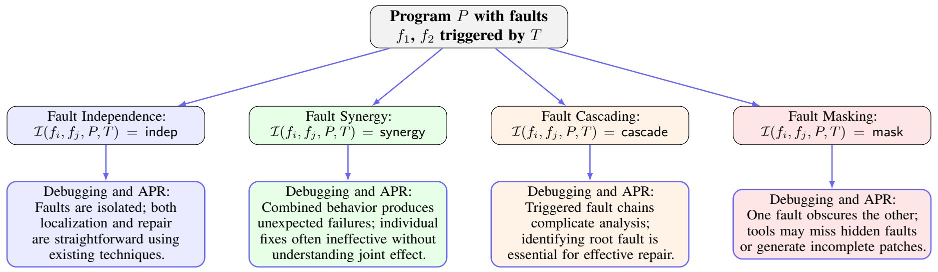
  Fault cascading occurs when one fault, f_i, triggers a chain reaction that activates another fault, f_j, which was previously dormant or unobservable. This interaction highlights the transitive nature of fault propagation, where f_i not only alters the execution path but also exposes f_j. Cascading complicates fault localization and repair, as f_i can cause f_j to manifest in previously hidden ways, making diagnosis and isolation harder. This underscores the importance of considering fault interdependencies in multi-fault programs, where one fault can have far-reaching effects on others.

D. Key Benefits of the Formal Model

The formal model of fault interactions we introduce provides a foundational understanding that offers several advantages for significantly enhancing current MFL and APR tools.

First, by formally articulating fault interaction types, our model provides a solid basis for designing and integrating interaction-aware program analyses into MFL and APR tools. For instance, understanding cascading effects (from our model) highlights the need for causal dependency graphs, while modeling masking and synergy informs the strategic use of techniques like mutation testing to reveal hidden behaviors.

Second, the model serves as a powerful lens for identifying critical gaps in the current literature regarding multi-fault scenarios. By providing a formal framework for fault interactions, it becomes significantly easier to pinpoint the key limitations of existing methods and articulate where foundational research is most urgently needed to address these complexities.



Program P with faults f_1, f_2 triggered by T
Fault Independence: $\mathcal{I}(f_i, f_j, P, T) = \mathsf{indep}$
Fault Synergy: $\mathcal{I}(f_i, f_j, P, T) = \mathsf{synergy}$
Fault Cascading: $\mathcal{I}(f_i, f_j, P, T) = \mathsf{cascade}$
Fault Masking: $\mathcal{I}(f_i, f_j, P, T) = \mathsf{mask}$
Debugging and APR: Faults are isolated; both localization and repair are straightforward using existing techniques.
Debugging and APR: Combined behavior produces unexpected failures; individual fixes often ineffective without understanding joint effect.
Debugging and APR: Triggered fault chains complicate analysis; identifying root fault is essential for effective repair.
Debugging and APR: One fault obscures the other; tools may miss hidden faults or generate incomplete patches.

Fig. 1. Fault interaction types and their impact on debugging and APR. $\mathcal{I}(f_i, f_j, P, T)$ denotes the semantic interaction between two faults under input T.

Finally, our model represents a significant step toward tackling two long-standing, interconnected challenges in APR: *patch overfitting* [4], [5] and *false positives* [6]. By explicitly modeling the nuanced interactions between multiple faults, it enables a clearer understanding of their collective impact on program behavior. This deeper insight is crucial for enhancing the trustworthiness of generated patches, helping to prevent patches that merely pass test cases while silently failing in more complex, real-world scenarios due to unaddressed fault interactions.

III. Implications for Debugging and Repair Tools

Each type of fault interaction introduces distinct challenges that can fundamentally mislead traditional debugging, fault localization, and automated program repair tools. By grounding our analysis in the formal model, we highlight these impacts.

- **Fault Localization (FL).** Traditional single-fault localization methods ($\mathsf{Loc}_{\text{single}}$) typically pinpoint a single most suspicious location based on observed failures. In multi-fault programs:
  - **Masking (M) & Synergy (S) Impact:** These interactions directly undermine $\mathsf{Loc}_{\text{single}}$. A masking fault ($f_i$) can cause $\mathsf{Loc}_{\text{single}}$ to exclusively identify f_i (due to its disruptive behavior), completely overlooking a masked fault (f_j) that is present but unobservable. Similarly, synergistic faults (f_i, f_j) might produce a failure that $\mathsf{Loc}_{\text{single}}$ incorrectly attributes to only one of them, or to an unrelated location, because the combined effect is unexpected.
  - **Cascading (C) Impact:** When f_i cascades to f_j, $\mathsf{Loc}_{\text{single}}$ might mistakenly localize f_j as the root cause, rather than the initial fault f_i that enabled f_j's activation. This leads to misdiagnosis and ineffective repair attempts.
  - **Multi-fault Localization (MFL):** Even recent MFL techniques $\mathsf{Loc}_{\text{multiple}}$ [7], [8], [9], [10], [11], though explicitly designed for multi-fault settings, often struggle in the presence of complex fault interactions. Their effectiveness degrades when assuming fault independence or applying simplistic aggregation of suspiciousness, missing the subtleties of masking, synergy, and cascading behaviors.
- **Test Suite Effectiveness.** The presence of fault interactions fundamentally challenges the notion of test suite adequacy:
  - **Masking (M) Impact:** A test suite might appear "adequate" if all known faults (or their initial manifestations) are covered. However, if masking faults are present, the test suite is *deceptively inadequate* as it fails to trigger masked faults. Patches based on such suites are prone to overfitting.
  - **Synergy (S) Impact:** For synergistic faults, individual tests targeting f_i or f_j might not reveal any failure, making the test suite seem adequate. Only specific tests that trigger their *combined* behavior will expose the bug, requiring targeted, interaction-aware test generation.
  - **Cascading (C) Impact:** Fixing f_i can alter execution paths, causing new failures from f_j. This means tests that previously passed might now fail, revealing test suite incompleteness only *after* a partial fix. Traditional test suites are not designed to dynamically adapt to these changes.

  Understanding fault interactions is crucial for designing more robust and evolving test suites that can unmask and expose a comprehensive view of program faults.
- **Automated Program Repair (APR) Stages.** The APR pipeline—from generation to validation—is deeply affected:
  - **Patch Generation (PG):** Most APR systems generate patches for faults *currently observed*. If a fault (f_j) is masked (f_i), it will not be identified by FL, and thus no patch will be generated for it. This inevitably leads to incomplete repairs. For synergistic faults, generating individual patches for f_i and f_j might not resolve the combined problem, or might even worsen it, necessitating joint patch generation.
  - **Patch Validation (PV):** This is where the core challenge of patch overfitting, amplified by interactions, emerges. A patch for f_i may pass all current tests, but if f_i was masking f_j, its application can *reveal* f_j, triggering new failures. Traditional PV would wrongly deem the f_i patch invalid (a false negative), discarding it despite being a correct fix.
  - **Repair Orchestration:** Without understanding interactions, the order of patch application can be disastrous (e.g., fixing a symptom (f_j in cascading) before the root cause (f_i)). This leads to unstable repair processes and wasted effort.

Fig. 1 visualizes the four interaction types and their impact on debugging and repair, reinforcing why traditional tools struggle in multi-fault contexts and motivating interaction-aware strategies.

IV. Making Multi-Fault Programs Debuggable

The formal model in Section II characterizes how fault interactions undermine existing debugging and repair approaches. We now turn to the practical question: how can this understanding be used to make multi-fault programs tractable for real-world tools? This section first surveys the current landscape of fault localization, automated program repair, and AI-based techniques to identify key limitations in the presence of interacting faults. We then present a roadmap for *interaction-aware* approaches that explicitly incorporate our model into both analysis and repair.

TABLE I
COMPARISON OF MFL AND AI TOOLS IN MULTI-FAULT SCENARIOS

Feature	**MFL tools**	**AI-based tools**
Example tools	CFaults	ChatGPT
Multi-fault support	Explicit	Implicit
Masking fault handling	Partial	Inconsistent
Synergistic faults	Unsupported	Occasionally
Cascading effects	Unsupported	Unsupported
Localization precision	High (indep.)	Variable
Patch generation	Not supported	Supported
Fault modeling	Explicit	Absent
Explainability	High	Low

TABLE II
COMPARISON OF APR AND AI TOOLS IN MULTI-FAULT SCENARIOS

Feature	**APR tools**	**AI-based tools**
Example tools	GenProg	ChatGPT
Fault assumption	Single-fault	Unspecified
Patch generation	Heuristic-based	Data-driven
Interaction awareness	Absent	Implicit
Test reliance	High	Medium
Patch generalization	Limited	Unpredictable
Overfitting risk	High	High
Explainability	Medium	Low
Adaptability	Limited	High

A. *Current Tool Landscape and Gaps*

Traditional FL techniques, such as spectrum-based methods, are primarily designed for *single-fault* scenarios. Recent advances, including tools like FLITSR [9] and CFAULTS [8], explicitly model multiple fault candidates and show promising performance when faults are independent. However, their effectiveness degrades in the presence of *fault interactions* that confound straightforward ranking or scoring strategies. This underscores a crucial shift: FL must evolve beyond assigning "suspiciousness" scores, towards reasoning explicitly about *causal chains* and *interdependencies* among faults.

Automated program repair (APR) techniques exhibit similar constraints. Widely used systems such as GENPROG [12], PROPHET [13], TBAR [14], and ANGELIX [15] operate under the single-fault assumption: once all failing tests pass, the program is deemed repaired. In multi-fault settings, this assumption breaks down—patching a visible fault can leave others untouched, expose previously masked defects, or worsen behavior if faults exhibit synergy. The outcome is often a *partially correct* patch that passes the current test suite but fails to generalize.

Emerging AI-based tools, including large language models (LLMs) such as CHATGPT [16] and CODET5 [17], bring new capabilities. They can localize or patch programs even in multi-fault scenarios, and in certain cases implicitly handle interactions like masking or cascading. However, their success is driven by pattern matching over vast training corpora rather than explicit reasoning about fault interactions, leading to unpredictable coverage, inconsistent behavior, and low explainability.

Tables I and II compare MFL, APR, and AI-based tools in multi-fault contexts. A clear structural gap emerges: MFL tools provide interpretability but lack generative repair; AI-based tools generate repairs but lack explicit fault interaction models. Bridging this gap demands *interaction-aware hybrid approaches*.

TABLE III
INTERACTION-AWARE DEBUGGING AND REPAIR STRATEGIES

Type	**Detection**	**Repair**
Masking	Enhance suspiciousness metrics with passing-test coverage to uncover hidden faults.	Employ iterative re-localization and patching to progressively reveal masked faults.
Synergy	Cluster code regions co-executed in failing tests to detect benign faults causing joint failures.	Generate joint patches addressing fault clusters, surpassing sequential single-fault fixes.
Cascading	Perform backward slicing from failure symptoms to identify root causes; construct causal dependency graphs.	Prioritize repairing initiating faults per causal graphs before fixing dependent faults.

B. *A Roadmap for Interaction-Aware Tools*

Our formal model captures three fundamental interaction types—*masking*, *synergy*, and *cascading*—and offers operational definitions to guide detection and repair. Table III summarizes how these definitions translate into targeted tool enhancements.

This roadmap points to a new class of *hybrid FL/APR pipelines* that combine MFL's interpretability and fault modeling with AI's generative adaptability. Structured MFL modules would identify interaction signatures via enhanced dynamic and static analysis, triggering specialized AI-based repair modules to generate joint patches or order repairs by causal dependencies. Grounding generative repair in explicit interaction models moves us from opportunistic fixes to *systematic, verifiable multi-fault debugging*.

Besides dynamic analysis, fault interactions may also be revealed through *mutation testing*, where undetected or jointly failing mutants provide evidence of masking or synergy. Likewise, *formal reasoning* offers a complementary route: by framing interaction types as verifiable properties (e.g., causal or relational dependencies), formal analysis tools could systematically check for their presence. While detailed exploration lies beyond our scope, these directions underscore that our model is compatible with both empirical and formal detection strategies

Integrating fault-interaction detection with repair strategies derived from our formal model enables our approach to transcend the prevalent assumption of fault independence. The resulting workflow systematically uncovers latent faults, resolves intricate multi-fault behaviours, and reframes programs once considered undebuggable as tractable cases for both research and practice.

V. CONCLUSION AND FUTURE WORK

In this paper, we introduced a formal model that defines and categorizes four fundamental types of fault interaction: independence, masking, synergy, and cascading. This framework reveals critical limitations in existing debugging and repair tools. Our analysis shows that without a principled understanding of these dependencies, current approaches risk misdiagnosis and incomplete repairs in complex multi-fault scenarios. The model also lays the groundwork for advancing fault localization, patch validation, and testing strategies. We urge the community to move beyond the single-fault assumption and develop scalable, fault-interaction-aware tools capable of addressing the inherent complexities of real-world multi-fault programs.

References

[1] A. Zakari, S. P. Lee, R. Abreu, B. H. Ahmed, and R. A. Rasheed, "Multiple fault localization of software programs: A systematic literature review," *Inf. Softw. Technol.*, vol. 124, p. 106312, 2020.

[2] 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.

[3] O. I. Al-Bataineh, "Current challenges in automated multi-fault program repair," in *IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE*. IEEE, 2025.

[4] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 532–543.

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

[6] H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *Empirical Software Engineering*, vol. 26, no. 20, pp. 1–25, 2021. [Online]. Available: https://link.springer.com/article/10.1007/s10664-020-09920-w

[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] 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.

[9] 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.

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

[11] R. Gao and W. E. Wong, "Mseer—an advanced technique for locating multiple bugs in parallel," *IEEE Transactions on Software Engineering*, vol. 45, no. 3, pp. 301–318, 2019.

[12] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," in *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, 2012, pp. 54–72.

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

[14] H. Tian, J. Jiang, Y. Xiong, X. Xie, L. Zhang, and H. Mei, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 155–166.

[15] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 691–701.

[16] OpenAI, "Gpt-4 technical report," https://openai.com/research/gpt-4, 2023, accessed: 2025-04-23.

[17] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2021, pp. 8696–8708.