Industrial Application of Deep Learning based Fault Localization with Mutation Features

Heechan Yang
School of Computing
KAIST
Daejeon, South Korea
heechan.yang@kaist.ac.kr

Ahcheong Lee
School of Computing
KAIST
Daejeon, South Korea
ahcheong.lee@kaist.ac.kr

Kyutae Cho
AI Research
LIG Nex1
Gyeonggi, South Korea
kyutae.cho2@lignex1.com

Yunsam Kim
R&D Dept.
VPlusLab Inc.
Gyeonggi, South Korea
yunsam.kim@vpluslab.kr

***Abstract*—Deep Learning-Based Fault Localization (DLFL) has achieved high fault localization accuracy, particularly by utilizing Mutation-Based Fault Localization (MBFL) features. However, the practical adoption of MBFL-based DLFL is severely hindered by two major hurdles: (1) the prohibitively huge computational costs of MBFL and (2) the absence of publicly available tools and configurations for systematic dataset construction. This paper describes the challenges and the authors' solutions to apply MBFL-based DLFL to industries, including South Korea's military defense LIG Nex1. Particularly, we have developed tools for the task and share our experience of systematically constructing the MBFL-based DLFL dataset, which is often ignored in the relevant research literature. Also, we provide concrete guidelines to configure the mutation experiment through our exploratory studies, which could reduce significant computational cost while keeping the high FL accuracy.**

With those aforementioned contributions, we have successfully applied MBFL-based DLFL to VPlusLab Inc.'s automated software testing tool (CROWN 2.0) and LIG Nex1's military defense systems. For example, we achieved high FL accuracy with a top-1 accuracy of 90.97% and reduced overall dataset construction costs by approximately 74.6% (from 7,508 to 1,907 CPU-hours) on the military defense systems.

***Index Terms*—deep learning-based fault localization, mutation-based fault localization, spectrum-based fault localization, automated debugging**

I. Introduction

The South Korean defense industry has solidified its position as a top-tier arms exporter, with massive contracts for the KF-21 Boramae fighter jet and K2 Black Panther tanks spanning from Europe to Southeast Asia. This rapid industrial expansion marks a fundamental shift from hardware-centric manufacturing to the development of software-defined combat systems. However, traditional validation efforts remain largely procedural, relying on manual testing and requirements traceability. Consequently, there is a need to integrate state-of-the-art debugging techniques such as Mutation-based Fault Localization (MBFL) and Deep Learning-based Fault Localization (DLFL).

Recent studies in DLFL have demonstrated promising results in achieving high fault localization accuracy [1]. This technique utilizes neural networks to learn complex patterns from various program artifacts. Specifically, these models leverage high-dimensional input features derived from both Spectrum-based Fault Localization (SBFL) and MBFL. By integrating these disparate data sources, DLFL captures intricate semantic relationships between program behavior and code structure, resulting in significantly higher accuracy in locating faults compared to traditional heuristic-based methods [2]–[10]. However, the efficacy of these models is heavily contingent upon the quality and scale of the underlying training dataset. In the context of real-world environments of the industries, the authors have found that the dataset construction process is often a major bottleneck. The dataset construction process can be very time-consuming and prone to human error, necessitating a more systematic and automated approach to data generation.

In the DLFL dataset, MBFL features play an important role in enhancing model accuracy. For example, the seminal paper on this approach [11] received the Most Influential Paper award in ICST 2024, recognizing its ability to capture the behavioral impact of individual code elements that traditional code coverage metrics often fail to detect [12].

However, the prohibitive computational overhead of the MBFL process during dataset generation remains a significant barrier for industry deployment. Although the performance of DLFL models depends significantly on MBFL features [12], [13], extracting MBFL features requires an exhaustive mutation analysis phase, often leading to excessive computational overhead. The process of generating, compiling, and executing millions of mutants is highly resource-intensive. For instance, constructing a comprehensive dataset for a military middleware system with approximately 10k lines of code can exceed 300 CPU-days (further discussed in Sect. VII-A2). Thus, such immense time costs prohibit industrial application of state-of-the-art automated fault localization (FL) techniques such as MBFL-based DLFL.

Another obstacle to industry deployment is the lack of publicly available tools for dataset construction. Currently, there is a critical absence of systematic frameworks tailored for DLFL data synthesis. Also, practitioners are forced to rely on ad-hoc parameter configurations due to a lack of a standard framework and related environment settings, leading to unpredictable and often suboptimal trade-offs between construction time cost and model performance.

To bridge the gap between academic research and industrial application, the authors at KAIST have maintained a

continuous collaboration for almost 10 years with VPlusLab Inc., an automated software testing and debugging firm, and LIG Nex1[1], a leading global defense exporter. Furthermore, KAIST, VPlusLab Inc., and LIG Nex1 formed a consortium to develop a unified framework to incorporate static analysis, automated testing, and automated debugging for the Ministry of Defense of South Korea, as a four-year long project. This partnership aims to facilitate the delivery and practical integration of state-of-the-art software testing and debugging techniques within the stringent operational constraints of the defense sector. The authors have applied state-of-the-art FL techniques (i.e., MBFL and DLFL) to commercial automated software testing tool CROWN 2.0 of VPlusLab Inc. [14] and defense middleware software systems of LIG Nex1 deployed in aerospace, maritime, and guided weapon applications. Also, we have transferred related tools as well as proper configurations of the experiments.

The main contributions of this paper are as follows:

1) In collaboration with VPlusLab Inc. and LIG Nex1, we developed a robust dataset construction framework for industrial applications.
2) To address industrial concerns regarding the massive computational overhead of MBFL and DLFL, we conducted systematic experiments to identify optimal configurations. This approach reduced dataset construction runtime cost by 74.6% compared to the naive configuration.
3) We evaluated our DLFL dataset construction framework on two industrial software systems: automated software testing tool CROWN 2.0 from VPlusLab Inc. and defense middleware software systems from LIG Nex1. Our results showed that the proposed methodology precisely localizes a real issue in CROWN 2.0 by achieving an average rank of 2.2 for the buggy function (Sect. VI-D) and top-1 function-level FL accuracy 90.97% on real-world defense systems (Sect. VII).

This paper is organized as follows. Sect. II reviews related FL techniques and examines their limitations on industrial applications. Sect. III provides a detailed account of the construction process for the DLFL training dataset. In Sect. IV and V, we present an exploratory study on MBFL configurations designed to optimize the effectiveness and efficiency of DLFL dataset generation. Building upon the insights gained from this study, we apply our methodology to two industrial software systems: CROWN 2.0 from VPlusLab Inc. (Sect. VI) and defense software from LIG Nex1 (Sect. VII). Finally, we present the lessons learned in Sect. VIII and offer concluding remarks in Sect. IX.

II. Related Works

A. Spectrum-Based Fault Localization (SBFL)

Spectrum-Based Fault Localization identifies faults by analyzing program spectra to assign suspiciousness scores to individual code elements [15]. Early formulas such as Tarantula [16], Ochiai [17], and DStar [18] demonstrated that statistical imbalances between passing and failing executions serve as a potent signal for FL. Later, advanced SBFL methods have evolved to incorporate genetic programming for the automated derivation of more effective formulas (GP13 [19]), or to integrate static code complexity metrics to refine traditional coverage-based signals (FLUCCS [20]). However, despite their computational efficiency, SBFL techniques remain inherently constrained by their reliance on execution frequency, which often fails to capture the semantic nuance necessary to diagnose complex software faults.

B. Mutation-Based Fault Localization (MBFL)

To overcome the limitations of SBFL, MBFL was introduced to evaluate the behavioral impact of specific code changes. Pioneer techniques such as Metallaxis [21] and MUSE [11] established the intuition that a faulty statement's signature is best captured through test outcome transitions, specifically Failing-to-Passing and Passing-to-Failing. By observing how artificial faults (mutants) alter the execution results of existing test cases, MBFL can more accurately pinpoint the root cause of a failure. However, while MBFL offers significantly higher accuracy, its practical application is often restricted by the massive computational overhead required to generate and execute millions of mutants.

C. Deep Learning-Based Fault Localization (DLFL)

The application of deep learning to FL has shifted the focus from manual heuristic design to automated representation learning. This evolution is characterized by diverse architectural paradigms designed to capture different aspects of program behavior. Convolution Neural Networks (CNNs), such as CNN-FL [9] and DeepRL4FL [10], treat coverage matrices as spatial patterns akin to image pixels, effectively identifying localized fault signatures within large matrices. Simultaneously, Graph Neural Networks (GNNs), exemplified by GRACE [6] and GNet4FL [5], have been employed to exploit the structural dependencies inherent in abstract syntax trees and dependency graphs. These models emphasize that the physical and logical structure of code is as important as its execution path.

Beyond architectural diversity, the focus of DLFL research has increasingly moved toward feature engineering and fusion. Models like DeepFL [12] and FusionFL [22] have proven that FL accuracy is maximized when a model integrates disparate data sources, such as textual similarity from information retrieval and semantic features from code representation learning. Advanced frameworks like MTL-TRANSFER [23] and CodeHealer [1] have further pushed the state-of-the-art by leveraging multi-task and transfer learning to generalize bug patterns across different software projects. However, a consistent trend in these studies is a focus on model complexity and feature depth, often at the expense of ignoring the underlying efficiency and effectiveness of the data construction process itself.

[1]The specific names of certain internal departments or project teams have been anonymized or kept confidential to comply with security protocols

D. Dataset Construction for DLFL with MBFL

Table I categorizes the DLFL techniques with MBFL features in the latest ACM survey paper published in December 2025 [24]. The table shows that most leading DLFL studies do not publicly disclose their mutant generation configurations or the specific MBFL implementation details used to construct their training datasets.

For instance, while state-of-the-art papers like CodeHealer [1] provide extensive details on their attention mechanisms, they offer no details for the parameter settings used to generate the underlying mutation data. Furthermore, several researchers modified standard mutation testing tools (i.e., PIT) to implement custom MBFL pipelines but have not disclosed these modifications [12]. While experienced researchers in fault localization may navigate these omissions, such omissions create a high barrier for industrial practitioners.

Also, most existing DLFL techniques incorporating MBFL features are limited to Java, making them incompatible with the typical C/C++ environments of defense software.

TABLE I: DLFL techniques incorporating MBFL features

Paper	Model	Lang.	Mutation tool	Mut. gen. conf. publicly available
DeepFL [12]	MLP	Java	PIT-1.1.5	X
GMBFL [33]	GNN	Java	Not specified	X
MTL-TRANSFER [23]	BiLSTM +MLP	Java	PIT-1.1.5	X
FusionFL [22]	LSTM+CNN +MLP	Java	PIT-1.1.5	X
CodeHealer [1]	MLP	Java	PIT-1.1.5	X
TRANSFER [27]	BiLSTM +MLP	Java	PIT-1.1.5	X
ALBFL [34]	transformer	Java	Not specified	X

E. Mutant Generation Tools

The efficacy and efficiency of MBFL are inherently tied to the capabilities of the underlying mutation engine.

PIT (PITest) [25] and Major [26] are popular mutation testing frameworks extensively utilized in Java-based FL research. However, since PIT was designed for test suite evaluation rather than fault localization, researchers [1], [12], [22], [23], [27] must modify its core engine to extract the execution signals required for MBFL. These custom modifications are rarely released to the public, creating a lack of transparency that obscures specific configuration details and hinders reproducibility. Major also has a scalability issue, such as build failures for generated mutants or system hangs when applied to large-scale projects [28]. Also, as both tools are strictly limited to the Java ecosystem, they are unsuitable for the C/C++ environments typical of high-reliability defense software.

MUSIC [29] is a robust mutation engine for C/C++ that operates at the source-code level, offering a comprehensive suite of mutation operators. In contrast, tools such as Mart [30], Mull [31], and SRCIROR [32] perform mutations at the LLVM Intermediate Representation (IR) level.

Those mutation tools at the IR level offer several practical advantages over source-level approaches:

1) **Simplified and robust operators**: The C++ grammar is notoriously complex, spanning hundreds of pages of evolving standards. Consequently, source-level mutation tools often struggle with type resolution and template logic, leading to a high percentage of stillborn mutants (those that fail to compile). For instance, a source-level operator change (e.g., replacing + with /) might accidentally target a custom operator+ function where the division operator does not exist. In contrast, LLVM IR uses a reduced instruction set where built-in arithmetic is clearly distinguished from function calls. This accuracy ensures that mutation operators are applied only where semantically valid, significantly reducing wasted computational effort on uncompilable mutants.
2) **Compilation efficiency**: Unlike traditional source-level mutation tools that require a separate compilation for each mutant, mutation tools in LLVM-IR level allow for a single, unified mutated binary, which requires only one compilation. By injecting conditional "mutant switches" directly into the IR, we can make a single mutated binary that contains thousands of mutants. Implementing this "all-in-one" approach at the source level is extremely difficult due to the syntactic complexity of C/C++. Wrapping mutation points in thousands of pre-processor macros or if-else blocks often interferes with template expansions and macro definitions, leading to frequent compilation failures.

Despite these benefits, we found these LLVM-based tools unsuitable for our industrial target software systems. Mull, for instance, is primarily optimized for mutation testing (i.e., calculating mutation scores) rather than the granular feature extraction required for MBFL. Furthermore, internal crash bugs and implementation constraints within Mart and SRCIROR, often related to handling complex industrial build systems, prevented their stable application.

Consequently, we utilized MUSIC for its robust source-level capabilities.

III. DLFL Dataset Construction Framework

A. Overview

The construction process, illustrated in Fig. 1, follows a structured pipeline designed to generate high-quality training data. The workflow is divided into two phases:

1) **Artificial bug insertion**: Developing an effective DLFL model requires a diverse repository of localization results. However, due to the sensitive nature of the industry (e.g., defense industry), real-world bug scenarios are often inaccessible or restricted. Also, the amount of real-world bug patches in industrial software may not be large enough for DLFL. To bridge this gap, we utilize mutation tools to inject artificial bugs into the target

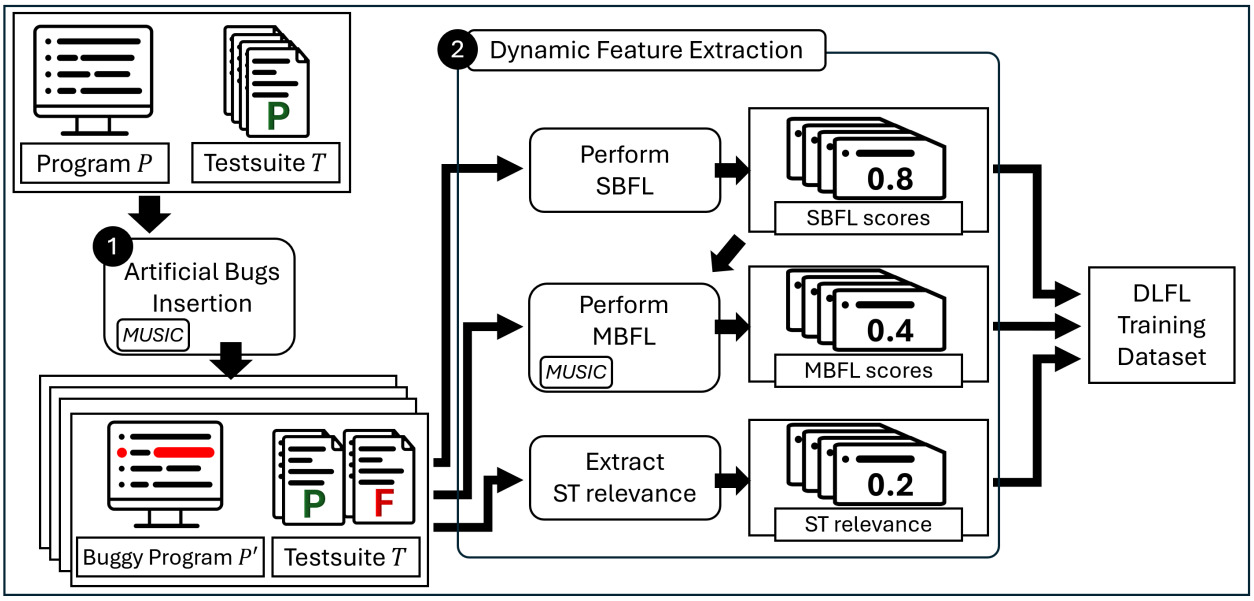

Fig. 1: Overall workflow of DLFL dataset construction, including bug insertion and feature extraction

software. This approach ensures a diverse set of fault scenarios (Sect. III-B).

2) **Dynamic feature extraction**: For each injected bug, we execute both SBFL and MBFL to collect baseline diagnostic data, and extract Stack Trace (ST) relevance to improve localization accuracy. (Sect. III-C).

B. Artificial Bug Insertion Phase

The efficacy of a DLFL model depends on its exposure to diverse fault patterns during training. However, our collaboration with LIG Nex1 highlighted a practical barrier: strict defense security protocols and confidentiality requirements prevent direct access to historical defect logs or real-world fault versions.

To address this, we employ mutants as a proxy for real-world defects [35], [36] (as illustrated in the left part of Fig. 1). We generated fault versions by injecting one mutant per code line across the entire project P using MUSIC. Each resulting mutant program is then validated against the existing test suite T. We retain only those mutants that cause at least one test failure while allowing other tests to pass. This approach ensures that each artificial bug provides a clear ground truth for the faulty line, enabling robust supervised learning [37].

By systematically injecting synthetic faults, we generate a comprehensive dataset that satisfies both the model's need for diversity and the project's stringent security constraints.

C. Dynamic Feature Extraction Phase

Once a set of artificial buggy programs, P', is established, the framework extracts a comprehensive set of nine dynamic features for every code line executed by a failing test case (illustrated in the right part of Fig. 1).

Following the methodology of CodeHealer [1], we incorporate six SBFL and two MBFL features. To further enhance localization accuracy, we supplement these with a Stack Trace (ST) relevance feature. The resulting nine-feature set is categorized as follows:

- **SBFL features:** six suspiciousness scores (DStar, GP13, Naish1, Naish2, Ochiai, and Tarantula) calculated from test coverage patterns.
- **MBFL features:** two suspiciousness values (MUSE and Metallaxis) are derived from subsequent mutation analysis. To generate mutants, we utilized MUSIC.
- **ST relevance feature:** A metric that quantifies the proximity of each statement to the runtime error context (i.e., the stack trace) as defined in [38]. Code elements that are closer to a point mentioned in the stack trace (e.g., file, function, line number) are assigned a higher value than those that are further away. This ensures that a code element is quantified by its proximal relationship to a crash.

The following Sect. III-D discusses the systematic approach to MBFL.

D. Systematic Optimization of MBFL Construction Cost

To balance localization effectiveness with computational efficiency, we first formalize the MBFL feature extraction process. We derive suspiciousness features from the generated artificial faults by applying a three-stage pipeline to the faulty program P and its test suite T:

1) **Mutation**: For each line in P executed by a failing test case, the framework mutates the line to generate mutants.
2) **Test execution**: Each mutant is executed against the test suite T. During this phase, we capture critical test result transitions, specifically identifying instances of Failing-to-Passing (failing tests that now pass) and Passing-to-Failing (passing tests that now fail).

3) **Metric calculation**: These transition signals are processed through standard MBFL formulas (i.e., MUSE and Metallaxis) to calculate the final suspiciousness values for each line.

Based on the process, we identify two key parameters that significantly impact both localization accuracy and computational overhead as follows:

1) **Mutation line selection ratio** (r): Generating mutants for every line executed by a failing test case is computationally prohibitive. To optimize this, we first rank and prioritize lines based on their SBFL scores (e.g., Ochiai). We then restrict mutation to the top r-percent of these ranked lines.
2) **Mutants per line** (m): Once target lines are selected, we determine the specific number of mutants (m) to generate for each line.

The total computational cost of the MBFL extraction process is linearly proportional to the product of these parameters ($r \times m$). Consequently, tuning these variables is essential for minimizing overhead without compromising the fidelity of the generated features.

To determine the optimal parameter values, we conducted an exploratory study using the popular Defects4J benchmark [39]. This benchmark provides a standardized environment to analyze the performance trade-offs of r and m before applying the model to the target industrial systems (Sect. IV and Sect. V).

E. Implementation

We developed DLFL training dataset generation framework in approximately 6,000 lines of Python code. Another contribution of our framework is the integration with an optimized version of the MUSIC [29]. We extended this engine to support selective generation of mutants based on a given line selection ratio (r) and mutation limit (m). The framework automates the entire process of mutant generation, program compilation, test execution, and feature extraction. This end-to-end automation makes the framework practical for real-world industrial use.

IV. Exploratory Study Setup

To ensure the techniques are viable for industry software, we first identify the optimal balance between computational efficiency and FL accuracy within a controlled environment.

A. Research Questions

To evaluate the trade-offs between computational overhead and FL accuracy, we address the following two research questions (RQs):

- **RQ-E1 (Target line selection ratio):** How does varying the selection ratio r (from 100% to 10%) impact the total dataset construction cost and the subsequent FL accuracy of the DLFL model?
- **RQ-E2 (Mutants per line):** To what extent does reducing the number of mutants per line m (from 10 down to one) influence computational overhead and the model's accuracy in FL?

TABLE II: Hyperparameters for DLFL model training

Batch	Epochs	Loss	Optimizer	LR	Dropout
64	40	Hinge-loss	Adam	10^{-3}	0.3

B. Subject Programs: Defects4J Benchmark

Following recent DLFL research papers [1], [22], [23], [27], we employed Defects4J (v1.2.0) [39] for evaluating automated FL. These projects collectively encompass 395 real-world faults, each accompanied by a comprehensive test suite and a confirmed ground-truth fix. Following the experiment setup of SmartFL [40], we excluded 133 bugs from the Closure compiler project in Defects4J.[2] We also excluded five faults that can not be reproduced in Java 11. Utilizing this benchmark allows an assessment of how our optimization parameters generalize across different programs.

C. DLFL Infrastructure

Building MBFL-based datasets is prohibitively expensive on single-node systems. Consequently, we deployed a distributed infrastructure consisting of 28 Linux machines. Each node is equipped with an AMD Ryzen 7 3800XT 8-core CPU and 32GB of RAM. To maximize hardware utilization during the mutation and execution phases, we executed 16 parallel processes on each node.

Feature extraction was managed automatically by our framework, integrating with PITest [25] for mutation analysis in Java. The framework collected coverage matrices for SBFL, performed transition analysis ($f2p$, $p2f$) for MBFL, and parsed failing test logs to compute the ST relevance scores for every record in the 257 fault instances.

The DLFL models were trained and evaluated on a workstation featuring an AMD Ryzen 9 5950X 16-core CPU and an NVIDIA GeForce RTX 4090 GPU.

D. Deep Learning Model Architecture and Hyper-Parameters

Following the experimental setup of CodeHealer [1], we implemented a simple Multi-Layer Perceptron (MLP) architecture consisting of a single hidden layer where the number of hidden units is the same number of nodes as the input layer (nine nodes, representing the SBFL, MBFL, and ST metrics). We also adopted the training hyperparameters from CodeHealer. The full configuration is detailed in Table II.

E. Evaluation Metrics

We evaluate the proposed methodology across two dimensions: computational efficiency and localization accuracy.

1) Computational Efficiency: Efficiency is measured as the elapsed time (CPU-hours) required for each stage of the construction process, including bug synthesis, feature extraction (SBFL, MBFL, and ST), and model training. This allows us to quantify the cost-reduction impact of the strategies investigated in RQ-E1 and RQ-E2.

[2]These bugs involve advanced language features that would require extensive modifications to the underlying mutation engine (PIT) to support.

TABLE III: Accuracy and cost trade-off analysis on Defects4J (RQ-E1). The mutant count is constant at 10 per line. The **bold** row (70%) indicates the optimal configuration.

Line sel. ratio	Mut. per line	Accuracy (%)			MFR
		Top-1	Top-3	Top-5	
100%	10	43.8	67.0	77.7	6.56
90%	10	43.8	67.5	77.6	6.42
80%	10	43.6	67.5	78.1	6.50
70%	**10**	**43.1**	**66.3**	**76.3**	**6.78**
60%	10	42.7	66.2	76.1	6.65
50%	10	41.7	66.2	76.0	6.72
40%	10	42.4	65.8	76.2	6.87
30%	10	40.4	65.3	75.3	7.11
20%	10	39.9	64.3	75.4	7.42
10%	10	40.1	63.9	74.3	7.59

2) Fault Localization Accuracy: Accuracy is evaluated at the function level, and we employ two primary metrics:

- **Top-N:** The number of faults where a faulty function appears within the top N positions of the ranked list. Higher top-N values indicate a higher success rate in practical settings.
- **Mean first rank (MFR):** The average rank assigned to the first faulty function of a bug. This serves as a direct proxy for the *debugging effort*, as it represents the number of functions a developer must inspect before finding the root cause (lower value is better).

For function-level FL, we employ a function-level aggregation based on the established methodologies [13], [20]. Each function f is assigned a suspiciousness score corresponding to the highest score among its statements. In cases where multiple functions share identical scores, we resolve these ties using the *max* tie-breaker method, where all tied functions are assigned the lowest possible (most conservative) rank.

3) Statistical Significance: To ensure the reliability of our findings and mitigate the risk of overfitting, we employed a 10-fold cross-validation repeated 10 times. In each iteration, the 257 faults were randomly partitioned into 10 folds; the model was trained on nine folds and evaluated on the held-out fold. We report the average top-N and MFR as results.

We use the Mann-Whitney U Test [41] with a significance level of $\alpha = 0.05$; a p-value below this threshold indicates that the observed difference is statistically significant.

V. Exploratory Study Results

A. RQ-E1: Impact of Target Line Selection Ratio (r)

Table III summarizes the results for RQ-E1, demonstrating a near-linear correlation between the selection ratio (r) and total processing time. Our analysis identifies $r = 70\%$ as the optimal threshold—the minimum ratio required to maintain statistical equivalence to the full (100%) baseline.

By restricting mutation to the top 70% of suspicious lines, we reduced dataset construction time by 29.8% (from 198.2 to 139.1 CPU-hours). Although the top-5 accuracy decreased slightly from 77.7% to 76.3%, a Mann-Whitney U test concluded that the performance difference is not statistically significant (p-value = 0.075). Conversely, reducing r below this 70% threshold resulted in a statistically inferior accuracy, confirming the 70% as the optimal configuration where efficiency is maximized without compromising diagnostic fidelity.

TABLE IV: Accuracy and cost trade-off analysis on Defects4J (RQ-E2). Line selection ratio is fixed at 70% (optimal from RQ-E1), except for the exhaustive baseline. **Bold** row (three mutants) indicates the optimal configuration.

Line sel. ratio	Mut. per line	Accuracy (%)			MFR
		Top-1	Top-3	Top-5	
100%	10	43.8	67.0	77.7	6.56
70%	10	43.1	66.3	76.3	6.78
70%	9	42.5	66.4	76.8	6.54
70%	8	42.5	65.9	76.3	6.46
70%	7	42.3	66.2	76.4	6.69
70%	6	42.6	66.2	76.7	6.67
70%	5	42.2	65.9	76.7	6.75
70%	4	42.5	65.9	76.5	6.79
70%	**3**	**42.4**	**66.7**	**76.2**	**6.93**
70%	2	42.3	65.6	76.4	6.90
70%	1	41.8	64.2	76.4	6.68

B. RQ-E2: Impact of Mutants Per Line (m)

Table IV shows the RQ-E2 results. Using the previously established selection ratio ($r = 70\%$), we systematically reduced the mutant density (m) from 10 down to one to identify the optimal configuration.

Reducing m from 10 to three slashed total processing time from 198.2 to 50.4 CPU-hours, a 74.6% cumulative reduction compared to the exhaustive baseline ($r = 100\%, m = 10$), and statistical validation confirms this configuration is viable (p-value = 0.079), which indicates no significant loss in accuracy. However, reducing m further to one or two resulted in a statistically inferior accuracy. Based on these findings, we establish the following cost-efficient mutation guideline: target the top 70% of lines with a density of three mutants per line.

VI. Industrial Application to CROWN 2.0: Case Study with VPlusLab Inc.

To evaluate the practical efficacy of the proposed DLFL dataset construction technique, we conducted a case study on CROWN 2.0, a commercial concolic unit testing tool developed by VPlusLab Inc.. Also, CROWN 2.0 has been applied to defense software of LIG Nex1 to improve software quality. This initial application served as a critical validation step prior to deploying the technique to the defense software.

Partnering with VPlusLab Inc. was strategically advantageous, as the firm possesses expertise in automated testing and debugging. This collaboration allowed the authors to refine the DLFL pipeline within an industrial environment, ensuring the methodology was robust and scalable before its application to the target defense systems.

TABLE V: Subject characteristics and bug injection statistics of industry software of VPlusLab Inc. and LIG Nex1

Company	Subject	Lang.	Size		Baseline line cov.	# Artif. Bugs	Avg. #FTs	Avg. #PTs	Avg. funcs by FTs	Avg. lines by FTs
			# funcs	# lines						
VPlusLab Inc.	CROWN 2.0	C++	159	2,204	59.7%	100	6.4	17.7	37.1	687.6
LIG Nex1	System_A	C	221	3,461	57.6%	50	2.5	74.4	100.6	1,328.8
	System_B	C	37	612	55.8%	50	1.4	11.6	19.1	298.9
	System_C	C	30	528	66.9%	50	3.6	7.1	15.9	229.1
	System_D	C++	442	2,226	60.9%	50	3.6	3.2	23.2	175.7
	System_E	C++	281	1,378	68.4%	50	3.2	31.9	78.1	330.3
	System_F	C++	305	1,472	45.8%	50	5.4	20.9	57.9	237.5

Our evaluation is structured to address one critical industrial requirement:

1) **RQ-C1 (Effectiveness on industry software)**: How effective is the framework's resulting dataset when applied to real-world industry software? (Sect. VI-B)
2) **RQ-C2 (MBFL feature contribution)**: To what extent does the fusion of MBFL features with SBFL and Stack Trace metrics enhance the accuracy of fault localization compared to a baseline excluding MBFL features? (Sect. VI-C)

Furthermore, we performed a case study on a real bug in CROWN 2.0 to demonstrate practical utility in localizing a non-trivial real bug within an industrial codebase (Sect. VI-D)

A. Experiment Setup

Due to the lack of real fault data for the CROWN 2.0 codebase, we synthesized a benchmark of 100 artificial bugs to facilitate a comparative analysis. To ensure this benchmark represents non-trivial debugging scenarios, we specifically filtered out bugs where the root cause and the crash site coincide, as such faults are easily revealed by stack traces of failing tests. Instead, we focused on synthesizing bugs where the original fault location is different from the point of failure.

We used the test suite written by CROWN 2.0 developers for our evaluation. As detailed in Table V, the artificial bugs resulted in an average of 6.4 failing tests (FTs) and 17.7 passing tests (PTs) per fault. Furthermore, the failing tests covered an average of 37.1 functions and 687.6 lines, providing the complex execution traces necessary for the model to learn latent error propagation patterns.

Following the generation of artificial bugs, we performed dynamic feature extraction—incorporating SBFL, MBFL, and ST relevance—to construct the DLFL training dataset. The evaluation was conducted through the following experimental configurations:

1) **RQ-C1 (Effectiveness on industry software)**: We generated a training dataset using MUSIC then performed a comparative analysis of the dataset construction time and the resulting fault localization accuracy to evaluate the performance of the MBFL-based dataset.
2) **RQ-C2 (MBFL feature contribution)**: To quantify the impact of mutation-based features, we constructed a baseline dataset that excludes MBFL features. We then compared the diagnostic accuracy of the DLFL models trained on each.

TABLE VI: Efficiency and accuracy of MBFL-based DLFL on CROWN 2.0

Category	Metric	MBFL-based DLFL
Wall clock time (h)		4.22
CPU-hours (MBFL)	Build	825.0
	TC Exec	268.9
	Total	1,093.3
DLFL acc.	Top-1	60.3%
	Top-3	82.1%
	Top-5	90.2%
	MFR	2.53

For the case study on fault localization of the real bug, we evaluated the practical utility of the models produced in RQ-C1 by applying them to a real-world bug scenario within the CROWN 2.0 codebase.

For these industrial applications, we adopted the optimal parameters identified in our exploratory study (Sect. V): a line selection ratio of $r = 70\%$ and a mutation limit of $m = 3$ mutants per line. For all experiments, we utilized the same hardware and deep learning model configuration detailed in the exploratory study setup (Sect. IV).

B. RQ-C1: Effectiveness of MBFL-based DLFL on industry software

Table VI summarizes the performance of MBFL-based DLFL on the CROWN 2.0 industrial software. Our framework achieved high fault localization accuracy, with Top-1 and Top-5 scores of 60.3% and 90.2%, respectively, effectively distinguishing faulty from non-faulty lines. In contrast, SBFL (Ochiai) yielded lower results: 41.0% (Top-1), 76.0% (Top-3), and an MFR of 3.35. Notably, the entire MBFL-based DLFL process was completed in 4.22 hours, highlighting its efficiency. These results demonstrate that our approach significantly outperforms traditional coverage-based methods while remaining computationally feasible for large-scale industrial applications.

1. run_crown crashes with the following code example. If you run the command "run_crown ./main 10 -dfs", the following error message will be shown:

```
Error: Sorts (_ BitVec 64) and (_ BitVec 32) are incompatible
```

```
#include<crown.h>
int main(){
  int a = -1;
  int b;
  SYM_int(b);
  a += 1 << (unsigned long)b;
  if(!a)
    return 0;
  return 1;
}
```

2. This bug happens in the line 127 of unary_expression.cc

Fig. 2: GitHub issue #27 report for CROWN 2.0

C. RQ-C2: Ablation Study on Mutation Features

Having established an efficient pipeline, we investigated the contribution of MBFL features to localization accuracy. To isolate the value of mutation-based information, we conducted an ablation study comparing the performance of a deep learning model trained on traditional features (Without MBFL) against a model augmented with mutation-based metrics (With MBFL).

As a result, the inclusion of MBFL features yields a substantial improvement in diagnostic precision, raising the top-1 accuracy from 40.0% to 60.3% (a 50.8% relative gain). Furthermore, the MFR decreased from 3.60 to 2.53. The Mann-Whitney U test confirms that the improvement is statistically significant (p-value=0.002). These results suggest that mutation-based features encapsulate latent semantic information—specifically regarding the behavioral changes of the program under perturbation—that is fundamentally absent in execution coverage or stack trace data alone.

D. Real Bug Case Study Discussion: Issue #27

We evaluated the DLFL model with our training dataset using a documented real-world crash from CROWN 2.0 (Issue #27). As illustrated in Fig. 2, the failure is derived from a bug in symbolic modeling of shift binary operations with mismatched operand bit-widths. While CROWN 2.0 is designed to translate target code expressions into formulas for the Z3 SMT solver [42], it failed to correctly process operations where the shift operands differed in size. Specifically, in the expression `1<<(unsigned long) b`, the left-hand operand is a 32-bit-sized integer, while the right-hand operand is a 64-bit-sized integer.

This error occurred because the implementation failed to account for the unique typing rules of the shift operator compared to other binary operators like addition and subtraction. While adding a 32-bit integer and a 64-bit integer typically results in a 64-bit integer, a shift operation on the same types should result in a 32-bit integer.

This issue is particularly challenging for traditional fault localization due to the distance between the root cause and the manifestation of the error. The system crashed within the unary expression parsing logic (`unary_expression.cc`) when handling a logical not operation (e.g., `!a`), but the actual defect was located in the binary expression parsing logic (`binary_expression.cc`). Because of this discrepancy, the traditional SBFL technique fails to accurately localize the fault, ranking the buggy function at 29.

In contrast, the DLFL model with our training dataset successfully located the buggy function. When trained using the MUSIC dataset, DLFL achieved an average rank of 2.2 for the buggy function. These results demonstrate the robustness of the training dataset constructed with our pipeline.

VII. Industrial Application to Defense Software: Case Study with LIG Nex1

We evaluate the practical applicability of our systematic DLFL dataset construction with defense software of LIG Nex1. By leveraging the insights gained from our previous evaluations on Defects4J and CROWN 2.0, we utilize the optimized configuration established in our previous studies (i.e., the 70% line selection ratio (r) and a mutation limit of $m = 3$).

A. Application on Defense Software Systems

We applied the proposed DLFL pipeline to six middleware software systems developed by LIG Nex1.

1) Subject Systems and Experimental Setup: The application was conducted on three C programs and three C++ programs, totaling approximately 10K lines of code. These systems are currently deployed in aerospace, maritime, and guided weapon applications. Due to strict defense security protocols (almost) prohibiting access to historical defect logs, we constructed an artificial bug dataset using 300 artificially injected faults (50 per system). The characteristics of these systems are provided in Table V. We utilized the test suite provided by LIG Nex1.

The experiments were conducted on an internal hardware setup at LIG Nex1, utilizing a 128-core Ubuntu server. The other experimental configuration remained consistent with the setup detailed in the CROWN 2.0 industrial application study (Sect. VI). We utilized MUSIC to construct the DLFL training dataset.

2) Accuracy and Efficiency Analysis: The model achieved high FL accuracy as summarized in Table VII. The optimized DLFL model reached a top-1 accuracy of 90.97%, top-3 accuracy of 98.00%, and top-5 accuracy of 98.97% (in function-level). Notably, MFR was recorded at 1.38, demonstrating that the buggy function is typically ranked within the first two suggested functions. For an industrial system comprising hundreds of functions, this level of FL accuracy suggests that developers can isolate root causes with minimal manual inspection, significantly reducing the debugging overhead.

TABLE VII: DLFL results on LIG Nex1 defense software

Top-1	Top-3	Top-5	MFR	CPU-hours	Wall-clock time taken
90.97%	98.00%	98.97%	1.38	1,907	15.3

Regarding computational efficiency, the adoption of optimized configurations ($r = 70\%, m = 3$) allowed us to reduce the high costs associated with exhaustive mutation analysis. The entire dataset construction process was completed in 15.3 hours of wall-clock time (1,907 CPU-hours) utilizing a 128-core parallel execution environment of LIG Nex1.

To quantify the impact of our optimization, we estimated the cost of an exhaustive analysis ($r = 100\%, m = 10$) based on the 74.6% mutation overhead reduction observed during our exploratory study (Sect. V-B). Applying this ratio to our industrial results, we estimate that the exhaustive approach would have exceeded 60.4 hours of wall-clock time (approximately 7,508 CPU-hours or 312 CPU-days). This reduction from 60.4 hours to 15.3 hours confirms that the optimized strategies identified in our research are essential for making MBFL-based DLFL feasible within the complex, large-scale C/C++ environments of the defense software domain.

Overall, the result indicates that the dataset construction strategy is not tied to a single benchmark. Even after moving from Defects4J and CROWN 2.0 to defense middleware systems with different code bases, the optimized configuration preserved strong localization accuracy.

VIII. Lessons Learned

A. Benefit of the DLFL Dataset Construction Framework

The application of our automated DLFL framework to the production software at VPlusLab Inc. and LIG Nex1 demonstrates the practical viability of deep learning-based debugging in industrial contexts. Our evaluation confirms the framework's high FL accuracy across two distinct scenarios in real-world industrial systems:

- Real-world bug scenario: successfully localized the faulty function of a documented system crash (i.e., the issue #27) at VPlusLab Inc. with an MFR of 2.2 (Sect. VI-D).
- Benchmark performance: achieved a top-1 accuracy of 90.97% on an artificial bug benchmark synthesized from defense software at LIG Nex1 (Sect. VII-A2).

Also, the framework resolves the bottleneck of computational cost. By applying our optimized feature extraction configurations, the total dataset construction time was reduced to 15.3 wall-clock hours (from 60.4 hours). This level of efficiency enables the practical integration of MBFL-based DLFL into industrial production cycles. We believe that this framework can significantly reduce manual debugging effort by prioritizing suspicious functions within a timeframe that accommodates delivery schedules.

B. Overcoming Industry's Inherent Insularity

The foremost challenge was overcoming the defense industry's inherent insularity. Persuading industrial partners to invest time and resources into new debugging techniques required long and extensive trust-building. Since 2017, the SWTV group at KAIST has cultivated a long-term partnership with LIG Nex1 to gain the necessary access to proprietary defense systems and published research papers together. For example, KAIST and LIG Nex1 established a LIG Nex1-KAIST Smart Defense 4.0 Research Center in 2018. This effort was further bolstered by VPlusLab Inc., an automated software testing firm with expertise in automated testing/debugging techniques for industrial applications.

C. Knowledge Transfer and Tool Application via On-site Collaboration

To facilitate the application of the DLFL framework into industry, the first author worked at LIG Nex1 full-time from June to August 2024. This full-time, on-site residency proved essential for bridging the gap between academic theory and industrial application, particularly for the following reasons:

- **Mitigating security constraints**: due to strict defense-sector security protocols, target source code cannot be accessed from outside of the company's secure internal network. Direct on-site access allowed the author to work with these sensitive codebases while complying with all non-disclosure and security requirements.
- **Rapid integration and refactoring**: Integrating a DLFL pipeline into a legacy industrial system often requires significant code refactoring to accommodate specific build systems and environment dependencies. Face-to-face collaboration with LIG Nex1 developers enabled a fast feedback loop, where technical issues were identified and resolved, accelerating the integration process.
- **Sophisticated technology transfer**: advanced FL techniques such as MBFL involve high conceptual complexity. In-depth on-site communication allowed the author to conduct deep-dive technical sessions, making these intricate mutation analysis strategies understandable and accessible to the company's engineers.

D. The Role of Ready-to-Use Tooling

As discussed in Sect. II-D, a significant barrier to the industrial adoption of DLFL is the lack of a deployment-ready tool. Most existing research tools are limited to prototypes that require practitioners to possess deep expertise in complex analysis, often necessitating extensive custom implementation to adapt to production environments.

To overcome this, we developed a comprehensive, automated pipeline for DLFL dataset construction. This "plug-and-play" tool significantly improved the efficiency of our collaboration with VPlusLab Inc. and LIG Nex1 by encapsulating intricate mutation logic and feature extraction into a standardized interface. By abstracting these technical complexities, we lowered the entry barrier for industrial practitioners, allowing them to utilize the framework effectively without needing to master underlying mechanics.

Furthermore, the ready-to-use nature of the tool accelerated deployment by enabling rapid integration into the company's

existing build environment. Providing a stable and documented toolset rather than just a methodology ensured that the technology remains actionable for industrial partners long after the initial research collaboration.

E. Necessity of Analysis-Friendly Testbeds

A critical, yet often overlooked, prerequisite for FL techniques is the ability to execute and observe each test case *separately*. In industrial settings, test suites are frequently designed as monolithic entities to perform high-level validation. At LIG Nex1, the existing infrastructure was built for aggregating the overall system stability, executing all test cases at once. While efficient for broad "pass/fail" reporting, this model is fundamentally incompatible with FL, which necessitates fine-grained execution to map specific coverage data to distinct program elements (Sect. II-A and II-B). To bridge this gap, the authors performed extensive refactoring of LIG Nex1's test suite, decoupling monolithic scripts to enable the per-test execution required for refined analysis.

Beyond the immediate technical refactoring, this experience suggests a broader challenge: the lack of standardized, analysis-oriented testing protocols in industrial workflows. While the decoupled testbed provided the necessary infrastructure, its long-term utility depends on how developers author future tests.

F. Industrial Feedback

Following the technology transfer and the technical seminar, we conducted a qualitative assessment with the lead researchers at LIG Nex1 who operated the delivered DLFL infrastructure. Their feedback provides important insights into the real-world utility of the system within the defense software domain.

One lead researcher highlighted the framework's ability to bridge the gap between theoretical models and proprietary systems:

> *"The face-to-face collaboration with the SWTV group of KAIST allowed us to successfully apply complex MBFL-based DLFL to the defense software of our company. ... Now we can localize a faulty function in a few hours, not in a few days, which is amazing!"*

This feedback shows that the systematic dataset construction methodology not only meets the rigorous accuracy requirements of military software but also aligns with the industrial need for automated, scalable debugging solutions.

IX. Conclusion

The deployment of MBFL-based DLFL has long been hindered by the prohibitive computational costs of constructing mutation-based datasets and the absence of a standard tool for dataset construction. These challenges serve as a barrier to its practical adoption in the military defense domain, where reliability is paramount but resources for exhaustive analysis are constrained. This paper has proposed, implemented, and validated a systematic methodology for MBFL-based DLFL dataset construction that effectively resolves the conflict between computational efficiency and FL accuracy in real-world industrial settings.

As future work, we plan to integrate state-of-the-art fault localization with advanced automated testing techniques [43], [44] to create a comprehensive testing and debugging framework. Ultimately, this integration will move us closer to a fully autonomous debugging pipeline capable of securing mission-critical systems against increasingly complex defects.

Acknowledgment

This research was supported by SW Research and Development Program(No. UC210018AD) and the National Research Foundation grants funded by the Korea government (RS-2021-NR060080 and NRF-RS-2024-00357348). The authors appreciate valuable discussions with Prof. Moonzoo Kim at KAIST/VPlusLab Inc. and Namhoon Jung at LIG Nex1.

References

[1] L. Zhang, S. Guo, Y. Guo, H. Li, Y. Chai, R. Chen, X. Li, and H. Jiang, "Context-based transfer learning for structuring fault localization and program repair automation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 4, Apr. 2025. [Online]. Available: https://doi.org/10.1145/3705302

[2] V. Csuvik, R. Aszmann, Beszédes, F. Horváth, and T. Gyimóthy, "On the stability and applicability of deep learning in fault localization," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024, pp. 546–555.

[3] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and X. Zhang, "A study of effectiveness of deep learning in locating real faults," *Information and Software Technology*, vol. 131, p. 106486, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920302287

[4] J. Yu, Y. Lei, H. Xie, L. Fu, and C. Liu, "Context-based cluster fault localization," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 482–493. [Online]. Available: https://doi.org/10.1145/3524610.3527891

[5] J. Qian, X. Ju, and X. Chen, "Gnet4fl: effective fault localization via graph convolutional neural network," *Automated Software Engg.*, vol. 30, no. 2, Apr. 2023. [Online]. Available: https://doi.org/10.1007/s10515-023-00383-z

[6] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 664–676. [Online]. Available: https://doi.org/10.1145/3468264.3468580

[7] X. Gou, A. Zhang, C. Wang, Y. Liu, X. Zhao, and S. Yang, "Software fault localization based on network spectrum and graph neural network," *IEEE Transactions on Reliability*, vol. 73, no. 4, pp. 1819–1833, 2024.

[8] X. Chen, T. Sun, D. Zhuang, D. Yu, H. Jiang, Z. Zhou, and S. Li, "Hetfl: Heterogeneous graph-based software fault localization," *IEEE Transactions on Software Engineering*, vol. 50, no. 11, pp. 2884–2905, 2024.

[9] Z. Zhang, Y. Lei, X. Mao, and P. Li, "Cnn-fl: An effective approach for localizing faults using convolutional neural networks," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 445–455.

[10] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, p. 661–673. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00067

[11] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 153–162.

[12] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 169–180. [Online]. Available: https://doi.org/10.1145/3293882.3330574

[13] Y. Kim, S. Mun, S. Yoo, and M. Kim, "Precise learn-to-rank fault localization using dynamic and static features of target programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3345628

[14] VPlusLab Inc., "Automated software testing solution CROWN 2.0," 2019. [Online]. Available: https://www.vpluslab.kr/en/solution

[15] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, p. 707–740, Aug. 2016. [Online]. Available: https://doi.org/10.1109/TSE.2016.2521368

[16] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:938359

[17] A. OCHIAI, "Zoogeographical studies on the soleoid fishes found in japan and its neighhouring regions-ii," *NIPPON SUISAN GAKKAISHI*, vol. 22, no. 9, pp. 526–530, 1957.

[18] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012, pp. 21–30.

[19] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "Human competitiveness of genetic programming in sbfl: Theoretical and empirical analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 1, pp. 4:1–4:30, Jul. 2017.

[20] J. Sohn and S. Yoo, "FLUCCS: Using code and change metrics to improve fault localisation," in *Proceedings of International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, Jul. 2017, pp. 273–283.

[21] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Softw. Test. Verif. Reliab.*, vol. 25, no. 5–7, p. 605–628, Aug. 2015. [Online]. Available: https://doi.org/10.1002/stvr.1509

[22] Y. Zhang, Y. Wang, D. Zhu, and W. Liu, "Fusionfl: A statement-level feature fusion based fault localization approach," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2024, pp. 37–46.

[23] X. Wang, H. Yu, X. Meng, H. Cao, H. Zhang, H. Sun, X. Liu, and C. Hu, "Mtl-transfer: Leveraging multi-task learning and transferred knowledge for improving fault localization and program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 6, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3654441

[24] C. Liu, Y. Lei, H. Xie, J. Wang, Y. Yu, and D. Lo, "Survey on learning-based dynamic fault localization: From traditional machine learning to large language models," *ACM Comput. Surv.*, Jan. 2026, just Accepted. [Online]. Available: https://doi.org/10.1145/3787202

[25] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 449–452. [Online]. Available: https://doi.org/10.1145/2931037.2948707

[26] R. Just, "The major mutation framework: efficient and scalable mutation analysis for java," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 433–436. [Online]. Available: https://doi.org/10.1145/2610384.2628053

[27] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1169–1180. [Online]. Available: https://doi.org/10.1145/3510003.3510147

[28] M. Delahaye and L. du Bousquet, "Selecting a software engineering tool: lessons learnt from mutation analysis," *Software: Practice and Experience*, vol. 45, no. 7, pp. 875–891, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2312

[29] D. L. Phan, Y. Kim, and M. Kim, "Music: Mutation analysis tool with high configurability and extensibility," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018, pp. 40–46.

[30] T. T. Chekam, M. Papadakis, and Y. Le Traon, "Mart: a mutant generation tool for llvm," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1080–1084. [Online]. Available: https://doi.org/10.1145/3338906.3341180

[31] A. Denisov and S. Pankevich, "Mull it over: Mutation testing based on llvm," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2018, pp. 25–31.

[32] F. Hariri and A. Shi, "Srciror: a toolset for mutation testing of c source code and llvm intermediate representation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 860–863. [Online]. Available: https://doi.org/10.1145/3238147.3240482

[33] S. Wu, Z. Li, Y. Liu, X. Chen, and M. Li, "Gmbfl: Optimizing mutation-based fault localization via graph representation," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 245–257.

[34] Y. Pan, X. Xiao, G. Hu, B. Zhang, Q. Li, and H. Zheng, "Albfl: A novel neural ranking model for software fault localization via combining static and dynamic features," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2020, pp. 785–792.

[35] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: challenging bug-finding tools with deep faults," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 224–234. [Online]. Available: https://doi.org/10.1145/3236024.3236084

[36] M. Pradel and K. Sen, "Deepbugs: a learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276517

[37] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 609–620. [Online]. Available: https://doi.org/10.1109/ICSE.2017.62

[38] H. Yang, "Systematic dataset construction for deep learning-based fault localization(dlfl) with mutation-based fault localization(mbfl) features," Master's thesis, KAIST (Korea Advanced Institute of Science and Technology), 2026.

[39] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[40] M. Zeng, Y. Wu, Z. Ye, Y. Xiong, X. Zhang, and L. Zhang, "Fault localization via efficient probabilistic modeling of program semantics," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 958–969. [Online]. Available: https://doi.org/10.1145/3510003.3510073

[41] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1–10. [Online]. Available: https://doi.org/10.1145/1985793.1985795

[42] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.

[43] A. Lee, I. Ariq, Y. Kim, and M. Kim, "Power: Program option-aware fuzzer for high bug detection ability," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2022, pp. 220–231.

[44] A. Lee, Y. Choi, S. Hong, Y. Kim, K. Cho, and M. Kim, "Zigzagfuzz: Interleaved fuzzing of program options and files," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–31, 2025.