석 사 학 위 논 문
Master's Thesis

변이 기반 결함 위치 추정(MBFL) 특성을 활용한 딥러닝 기반 결함 위치 추정(DLFL)용 체계적 데이터셋 구축

Systematic Dataset Construction for Deep Learning-Based Fault Localization(DLFL) with Mutation-Based Fault Localization(MBFL) Features

2026

양 희 찬 (梁 熙 讚 Yang, Heechan)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

변이 기반 결함 위치 추정(MBFL) 특성을 활용한 딥러닝 기반 결함 위치 추정(DLFL)용 체계적 데이터셋 구축

2026

양 희 찬

한 국 과 학 기 술 원

전산학부

변이 기반 결함 위치 추정(MBFL) 특성을 활용한 딥러닝 기반 결함 위치 추정(DLFL)용 체계적 데이터셋 구축

양 희 찬

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2025년 12월 8일

심사위원장 김 문 주 (인)

심 사 위 원 고 인 영 (인)

심 사 위 원 유 신 (인)

Systematic Dataset Construction for Deep Learning-Based Fault Localization(DLFL) with Mutation-Based Fault Localization(MBFL) Features

Heechan Yang

Advisor: Moonzoo Kim

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computing

Daejeon, Korea
December 8, 2025

Approved by

Moonzoo Kim
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS 양희찬. 변이 기반 결함 위치 추정(MBFL) 특성을 활용한 딥러닝 기반 결함 위치 추정(DLFL)용 체계적 데이터셋 구축. 전산학부 . 2026년. 51+v 쪽. 지도교수: 김문주. (영문 논문)
Heechan Yang. Systematic Dataset Construction for Deep Learning-Based Fault Localization(DLFL) with Mutation-Based Fault Localization(MBFL) Features. School of Computing . 2026. 51+v pages. Advisor: Moonzoo Kim. (Text in English)

초 록

딥러닝 기반 결함 위치 추정(Deep Learning-Based Fault Localization, 이하 DLFL)은 소프트웨어 디버깅 자동화 분야에서 높은 정확도를 거두고 있으며, 특히 변이 기반 결함 위치 추정(Mutation-Based Fault Localization, 이하 MBFL)의 특징을 결합하여 결함 탐지의 정밀도를 극대화하고 있다. 그러나 MBFL은 수많은 변이체(Mutant)를 생성하고 실행하는 과정에서 막대한 계산 비용(약 408 CPU-일 이상)이 발생할 뿐만 아니라, 이를 체계적으로 수행할 수 있는 공개된 데이터셋 구축 도구가 부재하다는 문제점이 있다. 이러한 비용적, 기술적 진입 장벽은 DLFL 기술의 실무적 적용을 저해하는 주요한 병목 현상으로 작용하고 있다.

본 연구는 이러한 한계를 극복하기 위해 MBFL 기반 DLFL의 실효성을 확보할 수 있는 데이터셋 구축 최적화 방법론을 제안한다. 특히, 자원 소모가 크고 복잡도가 높은 실제 국방 무기체계 소프트웨어에 최적화 기법을 직접 적용하기에 앞서, 통제된 벤치마크 환경인 Defects4J를 대상으로 탐색적 연구(Exploratory Study)를 선행하였다. 이 과정을 통해 데이터셋 구축 비용에 결정적인 영향을 미치는 두 가지 핵심 파라미터인 **(1) 변이체 생성 대상 라인 선택 비율**과 **(2) 라인당 변이체 생성 개수**에 대한 체계적인 실험적 분석을 수행하여, 성능 저하를 최소화하면서도 비용을 극대화하여 절감할 수 있는 최적의 가이드라인을 수립하였다. 또한, 개발자가 오류 분석 시 스택 트레이스(Stack Trace) 정보를 활용하는 디버깅 직관을 정량적으로 모델링한 **(3) 스택 트레이스 관련성 특징(Stack Trace Relevance Feature)**을 최초로 설계하여 딥러닝 모델의 새로운 학습 특징으로 제안하였다.

나아가 본 연구에서는 탐색적 연구를 통해 도출된 이론적 최적화 설정을 실무 현장에 효과적으로 이식하기 위해 약 6,000라인 규모의 파이썬 기반 데이터셋 자동 구축 도구를 개발하였다. 개발된 도구는 L사의 실제 국방 무기체계 DLFL 운용 시스템에 성공적으로 탑재 및 통합되었으며, 항공, 해양, 유도무기 등 실제 미들웨어로 구성된 6종의 국방 소프트웨어를 대상으로 실무 사례 연구를 수행하여 제안 기법의 실질적인 적용 가능성을 입증하였다. 최종적으로 실제 국방 소프트웨어 환경에서 **Top-5 기준 85%의 높은 정확도를 달성**함과 동시에 데이터셋 구축 비용을 약 **79% (9,081시간 → 1,907시간) 절감**함으로써, 본 연구가 제안하는 자동화 도구와 데이터셋 구축의 최적화 가이드라인이 DLFL의 실용성을 확보함과 동시에 결함 탐지 성능을 개선할 수 있음을 입증하였다.

핵 심 낱 말 딥러닝 기반 결함위치추정, 변이 기반 결함위치추정, 스펙트럼 기반 결함위치추정, 디버깅, 소프트웨어 테스팅

Abstract

Deep Learning-Based Fault Localization (DLFL) has achieved high accuracy performance in the field of software debugging automation, particularly by maximizing diagnostic precision through the integration of features from Mutation-Based Fault Localization (MBFL). However, the practical adoption of MBFL-based DLFL is severely hindered by two major hurdles: (1) the prohibitive computational costs of MBFL (exceeding 408 CPU-days) and (2) the critical absence of standardized, publicly available tools for systematic dataset construction. These economic and technical barriers serve as primary bottlenecks

for real-world application.

To overcome these limitations, this research proposes a systematic methodology for optimizing the mutation-based DLFL dataset construction process. Specifically, an Exploratory Study was first conducted on the Defects4J open-source benchmark to ensure the viability of optimization techniques for complex and high-cost military defense software. Through this exploratory study, foundational guidelines were established by analyzing two key parameters that govern construction costs: **(1) the target line selection ratio** and **(2) the number of mutants per line**. Furthermore, this thesis introduces a novel **(3) Stack Trace (ST) Relevance Feature**, which quantifies the diagnostic intuition of developers by modeling the context of runtime errors.

To efficiently conduct a systematic study of the dataset construction and enable practical applicability on military defense SW, we developed a comprehensive automation tool consisting of approximately 6,000 lines of Python code. This tool was successfully deployed and integrated into the operational DLFL infrastructure of Company L, a leading defense contractor. A practical case study was then conducted on six mission-critical C/C++ middleware systems (utilized in aerospace, maritime, and guided weapon systems) to validate the real-world applicability of the proposed methodology and the integrated infrastructure. In the actual military defense software environment, the proposed methodology achieved high performance with a **Top-5 accuracy of 85%** and **reduced overall dataset construction costs by approximately 79% (from 9,081 to 1,907 CPU-hours)**. These findings prove that the proposed dataset construction tool and optimization guidelines ensure the practical feasibility of DLFL while significantly advancing its diagnostic precision in high-reliability software domains.

Keywords deep learning-based fault localization, mutation-based fault localization, spectrum-based fault localization, debugging, software testing

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Fault Localization (FL) is the process of identifying the specific software elements, such as source files, functions, or lines of code, responsible for observed program failures. This task is widely recognized as one of the most resource-intensive and expensive phases of the software development life cycle. Traditionally, developers have relied on manual inspection, navigating through complex codebases and execution logs to pinpoint the root cause of a bug. This manual process is not only laborious and time-consuming but also highly prone to human error, especially as modern software systems continue to grow in size and complexity [3, 14]. The inherent difficulties of manual debugging have necessitated the development of automated fault localization techniques to improve software maintenance efficiency.

1.1 Background of Automated Fault Localization

Automated fault localization techniques aim to rank code statements by their likelihood of being faulty, allowing developers to prioritize their inspection efforts [41]. Early efforts led to the development of **Spectrum-Based Fault Localization (SBFL)**, which utilizes test coverage data from passing and failing test cases to rank suspicious code elements [47]. While computationally efficient, SBFL often lacks precision as it relies purely on coverage frequency. To address this, **Mutation-Based Fault Localization (MBFL)** was proposed, which injects artificial bugs (mutants) into the source code to observe how they affect test outcomes [25, 30]. Although MBFL captures deeper semantic relationships and achieves higher accuracy, its application is limited by the high costs of mutant execution.

More recently, **Deep Learning-Based Fault Localization (DLFL)** has emerged as a state-of-the-art solution. DLFL models utilize neural networks to learn complex patterns from various program artifacts. Specifically, these models leverage input features derived from both SBFL and MBFL to train the network. By integrating these disparate data sources, DLFL can capture intricate semantic relationships between program behavior and code structure, enabling significantly higher precision in locating faults compared to traditional heuristic-based methods [6, 52, 48, 34, 22, 9, 4, 51, 18]. However, as the field shifts toward these data-driven models, the practical challenges associated with constructing the requisite training datasets have become a primary concern for real-world deployment.

1.2 Problem Statement

Despite the high performance of DLFL in research settings, its practical adoption in industrial environments is severely hindered by two major technical and economic barriers [24, 39, 49, 50, 42, 17, 29]:

1. **High Computational Costs in Dataset Construction:** The effectiveness of DLFL relies heavily on MBFL features, which require an exhaustive mutation analysis phase. Generating and executing thousands of mutants to extract features is computationally expensive. For instance, building a dataset for a military middleware system with 61 KLoC can exceed 9,081 CPU-hours (approximately 408 CPU-days), making it impractical under tight development schedules.

2. **Lack of Standardized Public Dataset Construction Tools:** There is a critical absence of standardized, open-sourced tools for systematic DLFL dataset construction. Consequently, practi-

tioners are forced to rely on ad-hoc parameter settings, leading to unpredictable trade-offs between construction time and model performance.

These barriers have historically prevented the deployment of DLFL in mission-critical environments like Company L. Addressing these limitations through systematic optimization is essential to bridge the gap between academic advancements and industrial applicability.

1.3 Thesis Statement and Contributions

1.3.1 Thesis Statement

Prior DLFL research has focused heavily on improving model architectures while overlooking the systematic construction of training datasets. This thesis demonstrates that by optimizing dataset construction parameters and introducing stack trace relevance feature (runtime context features), it is possible to achieve high-performance fault localization that is both economically and technically viable for industrial defense software.

Thesis Statement

By systematically optimizing MBFL-based DLFL dataset construction and introducing a Stack Trace (ST) Relevance feature, (exploratory study of Defects4J) shows that it is possible to **(1) reduce dataset construction time by 74.6%** while maintaining accuracy equivalence and **(2) improve fault localization accuracy by up to 11.0%**. **(3) When applied to military defense software**, this methodology localized faults with high precision (**Top-5 85.0%**) while achieving a **79.0% reduction in construction costs**.

1.3.2 Key Contributions

The main contributions of this thesis are as follows:

1. **Systematic DLFL Dataset Construction Methodology:** This research conducts **the first comprehensive investigation** into the trade-offs between dataset construction efficiency and DLFL accuracy performance. Through an exploratory study on Defects4j, we establish empirical guidelines (70% line selection threshold and 3 mutants per line) that reduce construction time from by **74.6% (198.2 to 50.4 hours)** without sacrificing localization accuracy.

2. **Novel Stack Trace (ST) Relevance Feature:** We propose a novel approach to leveraging runtime stack trace information to capture the execution context of failures. By modeling the relevance of each code line to the runtime stack trace, this feature improves DLFL accuracy by **up to 11.0%** with negligible computational overhead.

3. **Application to Military Defense SW:** We developed a **6,000-line Python-based dataset construction tool** and successfully integrated it into Company L's operational DLFL infrastructure. Case studies on six mission-critical C/C++ systems demonstrate a **Top-5 accuracy of 85.0%** and a cost reduction of **79.0% (from 9,081 to 1,907 CPU-hours)**, proving the practical viability of the proposed methodology.

1.4 Structure of the Dissertation

The remainder of this dissertation is organized to reflect the transition from theoretical optimization to industrial application.

Chapter 2 (Background) establishes the technical foundations of Mutation-Based DLFL. It details the extraction mechanisms for Spectrum-Based (SBFL) and Mutation-Based (MBFL) features and describes the core deep learning model architecture, training protocols, and inference processes that serve as the baseline for this research.

Chapter 3 (Proposed Approach) presents our systematic methodology for reducing the computational costs of dataset construction. Furthermore, it introduces the primary technical contributions of this thesis: the mathematical formulation of the Stack Trace (ST) Relevance feature and the architecture of the developed MBFL-based DLFL dataset construction tool.

Chapter 4 and 5 (Exploratory Study) presents the preliminary investigation conducted on the Defects4J open-source benchmark. Since applying mutation analysis directly to military defense software is prohibitively expensive without prior optimization, these chapters establish foundational guidelines and identify optimal thresholds for line selection and mutant counts in a controlled environment.

Chapter 6 (Application to Military Defense Software) validates the proposed methodology and the dataset construction tool on real-world military defense C/C++ systems at Company L, followed by a discussion on its impact on the development process.

Chapter 7 (Related Work) reviews existing literature to position this study within the context of current DLFL research.

Finally, **Chapter 8 (Conclusion)** summarizes the dissertation and suggests future research directions.

Table 2.1: Formal Definitions of SBFL, MBFL, and Stack Trace Features for Dataset Construction

Feature Group	**Feature Name**	**List of Features**
SBFL	DStar [40]	$\frac{e_f(s)^2}{e_f(s)+n_f(s)}$
	GP13 [46]	$e_f(s) + \frac{e_f(s)}{2e_p(s)+e_f(s)}$
	Naish1 [26]	$-1\ if\ 0 < n_f(s),\ n_p(s)\ if\ 0 = n_f(S)$
	Naish2 [26]	$e_f(s) - \frac{e_p(s)}{e_p(s)+n_p(s)+1}$
	Ochiai [27]	$\frac{e_f(s)}{\sqrt{(e_f(s)+n_f(s))(e_f(s)+e_p(s))}}$
	Tarantula [11]	$e_f(s) + \frac{e_f(s)}{2e_p(s)+e_f(s)}$
MBFL	MUSE [25]	$\frac{1}{(\lvert\text{mut}(s)\rvert+1)(f2p+1)} \times \sum_{m\in\text{mut}(s)} (\lvert f_P(s) \cap p_m\rvert)$ $-\frac{1}{(\lvert\text{mut}(s)\rvert+1)(p2f+1)} \times \sum_{m\in\text{mut}(s)} (\lvert p_P(s) \cap f_m\rvert)$ where: • $mut(s)$ is the number of mutants generated on s. • $f_P(s)$ (or $p_P(s)$) is the set of tests that cover s and fail (or pass) on a target program P. • f_m (or p_m) is the set of tests that fail (or pass) on a mutant m. • $f2p$ (or $p2f$) is the number of test results that change from fail to pass (or pass to fail) for all mutants of P.
	Metallaxis [30]	$\max_{m\in\text{mut}_{\text{killed}}(s)}\left(\frac{\lvert f_P(s)\cap p_m\rvert}{\sqrt{totalFailing_{\text{TCs}}\times((\lvert f_P(s)\cap p_m\rvert)+(\lvert p_P(s)\cap f_m\rvert))}}\right)$ where: • $mut_{\text{killed}}(s)$: numbers of kill mutants on line s • $mut(s)$: set of mutants on line s
Stack Trace(ST)	ST Relevance (proposed method)	$\max_{f\in\text{stackTrace}(P,s)}\left(\frac{1}{position(f)+1} \times e^{-\lvert distance(s,f)\rvert^2}\right)$ where: • P: the faulty program version • $stackTrace(P,s)$: set of all stack frames f where line s appears from stack traces of failing tests on P • $position(f)$: 0-based position of stack frame f in stack trace (0 = top of stack, closest to failure) • $distance(s,f)$: difference between line number of s and line number in stack frame f

Chapter 2. Background

This chapter establishes the technical foundations of Mutation-Based Deep Learning-Based Fault Localization (MBFL-based DLFL). We provide a comprehensive overview of the DLFL pipeline, ranging from the structural components of the training dataset to the architecture of the neural networks used for localization. Specifically, we detail the extraction mechanisms of Spectrum-Based Fault Localization (SBFL) and Mutation-Based Fault Localization (MBFL) features, which represent the program's dynamic execution behavior. Finally, we describe the deep learning model training and inference process, providing the baseline configuration against which our proposed optimizations and novel feature are evaluated.

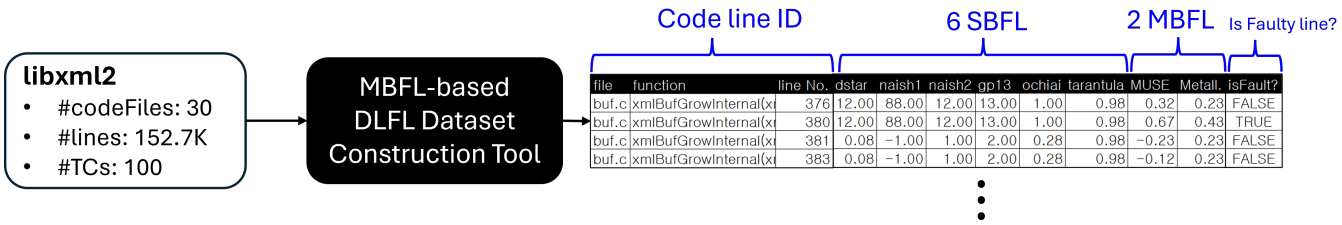

Figure 2.1: Overview of the MBFL-based DLFL dataset construction process and the resulting structured records used for model training.

2.1 Components of MBFL-based DLFL Dataset

In past frameworks of MBFL-based DLFL, the training dataset is structured as a collection of records, where each record represents a single source code line. Each record typically consists of 8 dynamic features, comprising 6 SBFL metrics and 2 MBFL metrics. These features serve as the primary input for the neural network to differentiate between faulty and non-faulty statements.

As illustrated in Figure 2.1, the construction process yields a dataset where each row corresponds to a specific code line executed by at least one failing test case. For every line, the 8 features provide quantitative suspiciousness scores (e.g., Ochiai, MUSE). Each record is further annotated with a ground-truth label (1 for faulty, 0 for benign). This structured format allows the model to perform a line-by-line classification or ranking task. The following sections describe how these fundamental features are calculated through dynamic program analysis.

2.1.1 Spectrum-Based Fault Localization (SBFL) Features

Spectrum-Based Fault Localization (SBFL) ranks suspicious code elements by analyzing the correlation between statement execution and test outcomes [41, 40, 47, 37, 15, 8]. To calculate SBFL features, we collect four fundamental spectral elements for each code line s:

- $e_p(s)$: The number of passing test cases executing the line s.
- $n_p(s)$: The number of passing test cases not executing the line s.
- $e_f(s)$: The number of failing test cases executing the line s.
- $n_f(s)$: The number of failing test cases not executing the line s.

By substituting these spectra into various formulas such as DStar [40], GP13 [46], Naish1 [26], Naish2 [26], Ochiai [27], and Tarantula [11], we generate the first 6 features of our dataset.

Figure 2.2 provides a concrete example illustrating this extraction. In this scenario, test case tc_2, which fails, uniquely covers line 5. Because line 5 is distinctly executed by a failing test and is not executed by any passing tests, it receives a high suspiciousness score according to metrics like Naish2. This highlights how SBFL leverages the statistical imbalance between failing and passing executions to isolate suspect code.

However, SBFL relies on the assumption of a direct correlation between execution frequency and faultiness. This assumed heuristic can be misled by coincidentally correct test cases or shared code blocks like initialization routines. These inherent limitations in precision necessitate the use of more semantic-aware techniques like Mutation-Based Fault Localization (MBFL).

void abs(**int** a, **int** b)	Coverage of Test Cases (a, b)		$e_p(s)$	$n_p(s)$	$e_f(s)$	$n_f(s)$	*Naish2 Susp.*
	tc_1: (1,2)	tc_2 (2,2)					
1: **if** (a>b)	•	•	1	0	1	0	0.5
2: return a-b;			0	1	0	1	0.0
3: **else if** (a<b)	•	•	1	0	1	0	0.5
4: return b-a;	•		1	0	0	1	-0.5
5: **return** 1; // fault		•	0	1	1	0	**1.0**
Test Results	Pass	Fail					

Figure 2.2: Concrete example of SBFL feature extraction based on coverage matrices.

2.1.2 Mutation-Based Fault Localization (MBFL) Features

Mutation-Based Fault Localization (MBFL) extends coverage-based analysis by evaluating the behavioral impact of artificial code changes, known as mutants. By observing how syntactic alterations affect the pass/fail status of tests, MBFL captures deep latent relationships between code elements and failures [25, 30].

The extraction of MBFL features involves a multi-stage process: first, target source code lines are selected; second, a set of mutants is generated for each selected line; and third, the entire test suite is executed for every individual mutant. During this testing phase, we track two critical test outcome transitions for each mutated line s:

- **Failing-to-Passing (f2p):** The number of tests that failed on the original program but pass on the mutant. A high $f2p$ count strongly suggests that the mutation partially corrected a fault at line s.
- **Passing-to-Failing (p2f):** The number of tests that passed on the original program but fail on the mutant. A high $p2f$ count indicates that the mutation broke a correct logic at line s.

These raw transition counts are then substituted into specific MBFL formulas, such as **MUSE** [25] and **Metallaxis** [30], to calculate the final 2 features of the dataset. For instance, the MUSE formula weighs $f2p$ transitions against $p2f$ transitions to normalize the suspiciousness score.

As shown in Figure 2.3, the granularity of MBFL provides superior diagnostic power. However, the requirement to generate and test numerous mutants for a wide range of selected lines introduces a massive computational burden. In large-scale systems, this exhaustive extraction process becomes the primary bottleneck for practical DLFL adoption.

2.2 Deep Learning Model Training and Inference

After constructing the DLFL dataset, deep learning models are trained with the extracted 8 dynamic features. This enables the model to distinguish between faulty and non-faulty lines effectively based on the combined signals of 6 SBFL- and 2 MBFL-based feaure values. In the inference phase, the trained model calculates suspiciousness scores (at a range of $[0.0, 1.0]$) for each code line, facilitating accurate fault localization by ranking lines from most to least suspicious.

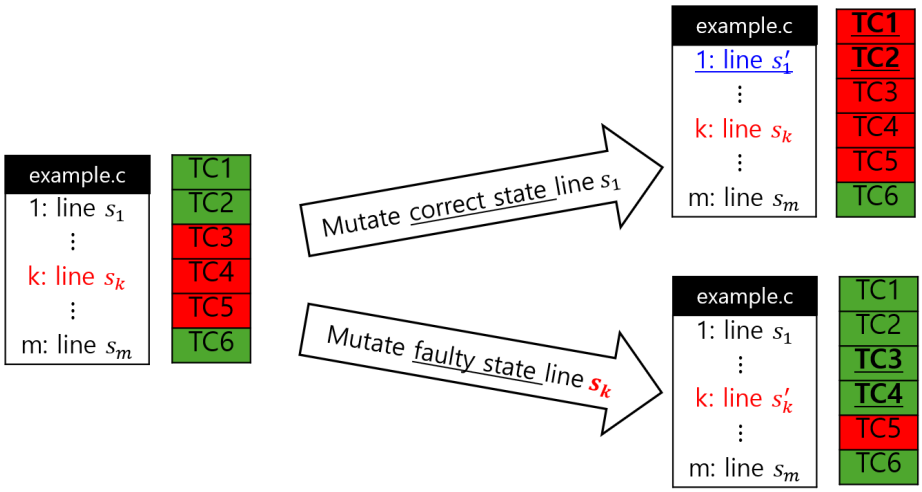

Figure 2.3: The MBFL process involving mutant generation and tracking of outcome changes for feature calculation.

2.2.1 Model Choice and Architecture

Following the CodeHealer [49] methodology, current state-of-the-art method on MBFL-based DLFL, a Multi-Layer Perceptron (MLP) is adopted as the core model for this study. This choice is motivated by the MLP's proven success in handling structured tabular input and its relative simplicity compared to more complex architectures. The model consists of one hidden layer with the same number of nodes as the input layer, utilizing fully connected layers followed by non-linear activation functions to learn the non-linear relationship between features and faultiness.

As shown in Figure 2.4, for each code line in the dataset, the input to the model is a 8-dimensional vector representing the features extracted from dynamic analysis. The model's input and output can be summarized as follows:

- **Input:** A 8-dimensional feature vector for each code line, composed of 6 SBFL formulas and 2 MBFL formulas.
- **Output:** A real-valued probability in the range $[0.0, 1.0]$, which is interpreted as the suspiciousness score of the corresponding code line.

During training, ground truth labels are assigned as 1 for faulty lines and 0 for benign lines. The model is optimized to maximize its ability to discriminate between these two classes. This MLP-based approach provides a straightforward and computationally efficient baseline for evaluating the impact of dataset construction decisions on overall DLFL performance.

2.3 Summary

In this chapter, we have reviewed the dataset components, extraction methods, and the deep learning model architecture established in DLFL research. We described how 8 dynamic features provide complementary signals by substituting raw execution data into well-defined formulas. Crucially, we highlighted how the exhaustive nature of MBFL extraction creates a significant computational bottleneck (specifically the selection of lines and the generation of mutants). In the following chapter, we present our

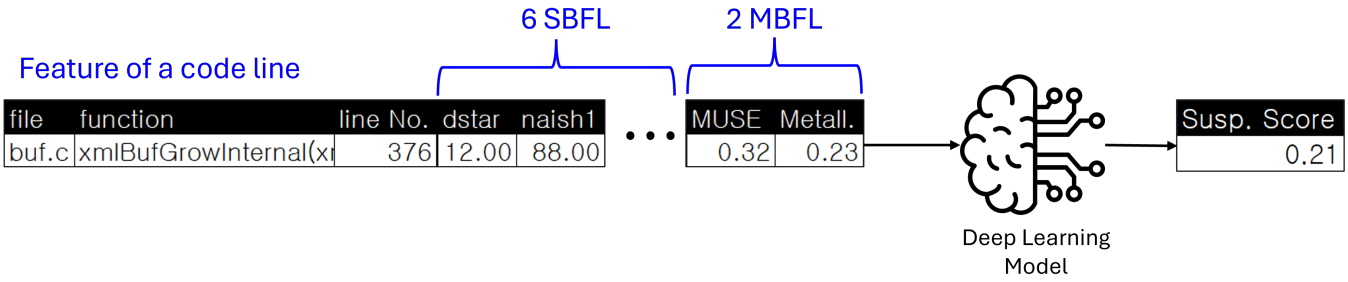

Figure 2.4: Architecture of the DLFL model and the mapping from dynamic features to suspiciousness scores.

proposed approach, which introduces a systematic methodology to optimize these MBFL parameters and a novel Stack Trace Relevance feature to further enhance localization accuracy.

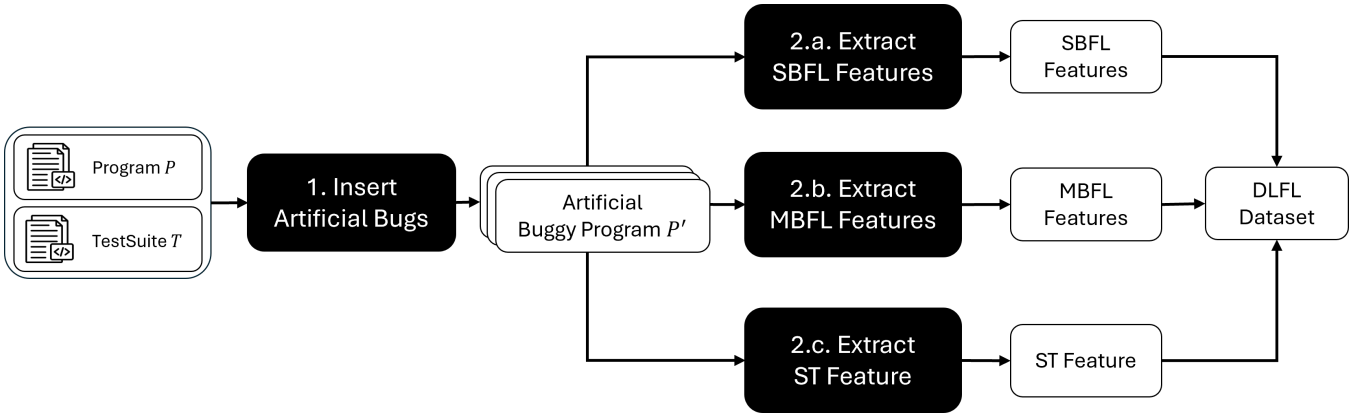

Figure 3.1: Overall workflow of MBFL-based DLFL dataset construction, including bug synthesis and feature extraction.

Chapter 3. Approach

This chapter presents a systematic methodology designed to transition Deep Learning-Based Fault Localization (DLFL) from a theoretical technique to a practically viable industrial solution. Our approach is driven by two major research objectives: (1) drastically reducing the prohibitive computational costs of mutation-based dataset construction and (2) significantly enhancing the fault localization accuracy of DLFL models through runtime error (stack trace) context analysis.

To achieve these goals, we first describe the end-to-end DLFL dataset construction workflow, accounting for the unique data accessibility constraints inherent in the defense industry. We then provide a granular analysis of the mutation phase to identify and optimize the key parameters governing construction time. Finally, we introduce the formal definition and motivation of the novel Stack Trace (ST) Relevance feature. All proposed strategies were implemented and validated using a custom-developed automation tool consisting of approximately 6,000 lines of Python code.

3.1 DLFL Dataset Construction Workflow Overview

The construction of a robust DLFL dataset involves transforming raw program artifacts into structured numerical records. As illustrated in Figure 3.1, the process proceeds through the insertion of artificial bugs followed by a comprehensive dynamic feature extraction phase.

3.1.1 Artificial Bug Generation

The performance of a DLFL model is highly contingent upon its exposure to diverse and representative fault patterns during the training phase. While the use of historical bug databases is ideal for research purposes, our collaboration with Company L revealed a critical practical barrier. Due to strict defense security protocols and the confidential nature of military software development, direct access to historical fault versions or actual defect logs was restricted.

To circumvent this limitation, we employ **Artificial Bug Generation** via mutation [36, 33]. The systematic process of bug insertion is illustrated in Figure 3.2. For the target defense software middleware, we generated fault versions by injecting one mutant per code line across the entire project P. Each

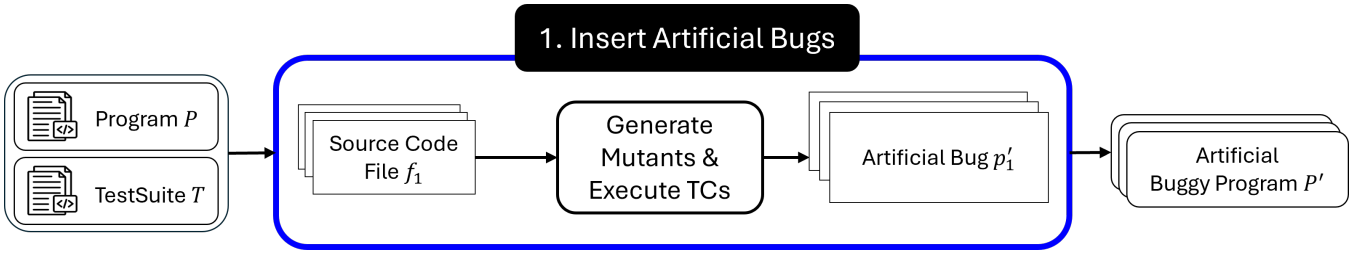

Figure 3.2: Process of artificial bug insertion to construct faulty program versions.

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 strict selection criterion ensures that each artificial bug provides a clear ground truth for the faulty line, enabling robust supervised learning even in environments where real-world failure data is unavailable [31].

3.1.2 Dynamic Feature Extraction Phase

Once a set of artificial buggy programs P_i' is established, the construction tool enters the dynamic extraction phase. For every code line exercised by a failing test case, the tool extracts a comprehensive set of 9 dynamic features. This marks an evolution from traditional DLFL datasets, which typically rely on 8 features (6 SBFL and 2 MBFL) as described in Chapter 2. By incorporating the newly proposed Stack Trace Relevance feature as the 9th component, we provide the model with critical runtime context. The full feature set comprises:

- **SBFL Features:** 6 suspiciousness values (DStar, GP13, Naish1, Naish2, Ochiai, Tarantula) are calculated based on test coverage patterns (e_p, e_f, n_p, n_f), as detailed in Section 2.1.1.
- **MBFL Features:** 2 suspiciousness values (MUSE, Metallaxis) are derived from subsequent mutation analysis ($f2p$, $p2f$) of the faulty program, as detailed in Section 2.1.2,.
- **ST Relevance Feature:** A novel feature proposed in this study that quantifies the proximity of each line to the runtime error context (stack trace).

The dynamic feature extraction phase represents the most computationally intensive stage of the entire lifecycle. While SBFL features are lightweight, the exhaustive nature of MBFL feature extraction introduces a massive temporal bottleneck that threatens the feasibility of the overall system. The following sections provide a detailed analysis of this cost-heavy phase to identify optimization points.

3.2 Systematic Optimization of MBFL Construction Cost

Mutation-Based Fault Localization (MBFL) features are essential for capturing latent semantic signals, yet their extraction is the primary driver of computational overhead. To address this, we formalize the extraction process to identify and optimize the key parameters that dictate construction time.

3.2.1 Identification of Key Parameters via MBFL Feature Extraction Process Analysis

The MBFL feature extraction process is a nested operation as illustrated in Figure 3.3. This phase captures how subtle code changes affect test outcomes, providing the deep diagnostic information required for high-precision models.

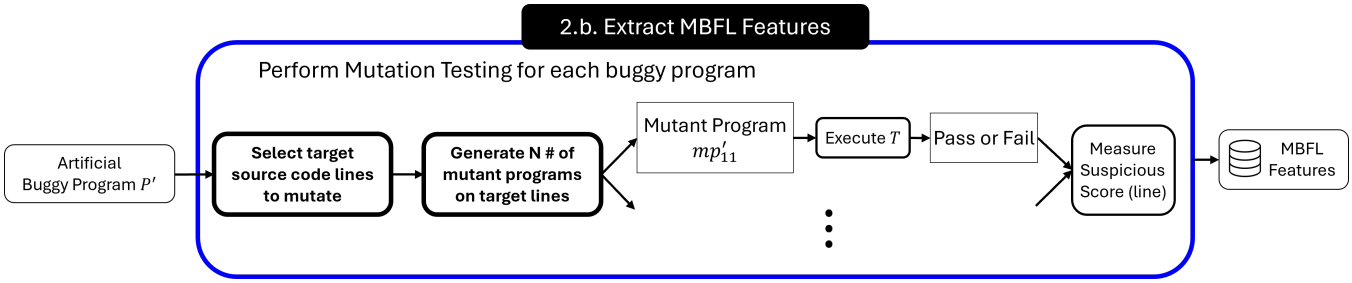

Figure 3.3: Detailed process of MBFL feature extraction highlighting the mutation and test execution bottlenecks.

The extraction workflow follows a rigid sequence of operations. First, **Target Line Selection** determines which suspicious code lines, ranked by initial SBFL scores, will undergo further analysis. Second, a specific number of **Mutant Programs are Generated** for each selected line to create semantic variations. Third, **Mutation Testing** is performed by executing the test suite against every mutant to capture transition signals such as Failing-to-Passing (f2p) and Passing-to-Failing (p2f). Finally, these signals are substituted into formulas such as MUSE and Metallaxis to calculate the final features.

Numerous studies have sought to reduce the computational cost of MBFL [38, 23, 19], proposing techniques like DMES [21] or SMBFL [2]. However, these methods have not been extensively explored in the context of large-scale MBFL-based DLFL dataset construction, where the goal is to build a training set rather than localize a single bug. By formalizing this workflow in Algorithm 1, we identify two primary parameters that control this complexity: the **Target Line Selection Ratio (r)** and the **Mutant Count Per Line (m)**. The systematic optimization of these parameters, which will be described in more detail in the following section, is the key to making DLFL viable for industrial systems.

3.2.2 Systematic Optimization Approach

Our strategy involves a systematic investigation into the trade-off between reduction and accuracy to find the optimal parameter configurations where computational cost is minimized while DLFL model performance remains statistically equivalent to the baseline configurations ($r = 100$, $m = 10$).

Optimization of Target Line Selection Ratio (r): Determining the precise percentage of suspicious lines to select is critical for scalability. While mutating all executed lines ensures maximum data availability, it leads to an explosion of redundant test executions. We evaluate the impact of restricting r to the top r-percent of lines ranked by Ochiai scores. By identifying a threshold that discards clearly benign lines while retaining all faulty statements for the mutation phase, we can prune the search space by orders of magnitude without affecting the final model quality.

Optimization of Mutant Count Per Line (m): Once target lines are prioritized, we must determine the minimum number of mutants (m) required per line. While a higher m provides more diverse transition data, it directly scales the compilation and test execution time. We systematically reduce m from an exhaustive set (e.g., 10 mutants) down to a minimal set to find the smallest sample size that allows the DLFL model to maintain statistical equivalence in localization performance. This density reduction is vital for environments with limited CPU resources.

Algorithm 1: MBFL Feature Extraction with optimization parameters r and m.

Input: P : target program, T : set of test cases, S : SBFL metric (e.g., Naish2), r : line selection ratio, m : number of mutants per line

Output: MBFL dataset

```
failing_tcs ← get_failing_tcs(T)
candidate_lines ← get_lines_covered(failing_tcs)
suspicious_lines ← order_by_sbfl_score(candidate_lines, S)
target_lines ← select_top_r_percent(suspicious_lines, r) ;      // Decision Parameter 1:
target line ratio
foreach line ∈ target_lines do
    mutants ← generate_mutants(line, m) ;      // Decision Parameter 2: number of
    mutants per line
    foreach mutant ∈ mutants do
        covering_tcs ← get_covering_test_cases(T, line)
        foreach tc ∈ covering_tcs do
            outcome ← execute_test_case(mutant, tc)
            save_result(line, mutant, tc, outcome)
```

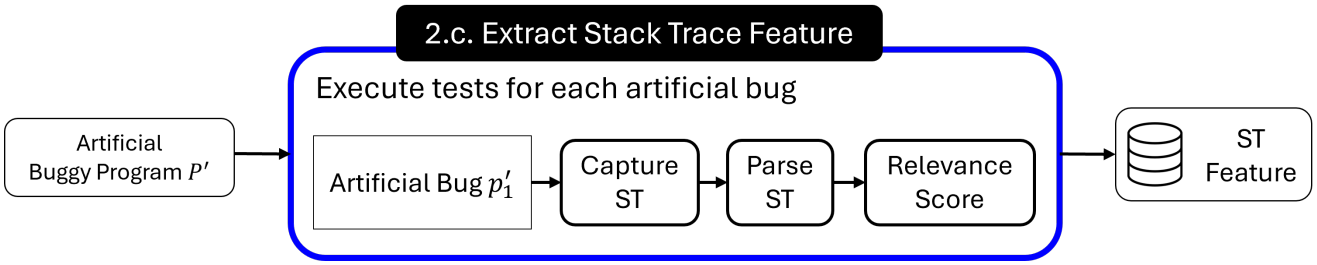

Figure 3.4: Process for extracting and formulating the Stack Trace Relevance feature.

3.3 Improving Localization Precision via Stack Trace Relevance Feature

The second major objective of this thesis is to enhance the fault localization accuracy of DLFL models. To achieve this, we propose the **Stack Trace (ST) Relevance feature**, a novel technical contribution that quantifies the structural and hierarchical relationship between source code lines and observed failures.

3.3.1 Motivation: Emulating Developer Diagnostic Heuristics

The ST Relevance feature is designed to formalize the fundamental debugging habits of human developers. When a program crashes, developers instinctively prioritize the stack trace, operating under the heuristic that code elements residing in the upper frames (closest to the point of program termination) are most likely to contain the root cause. Our proposed feature provides a diagnostic signal by directing the model's attention to the narrow, active path of the crash. By providing this runtime failure context, we enable the deep learning model to differentiate between lines that are merely executed and those that

are structurally relevant to the failure.

Example Scenario: Defects4J Lang-37b To illustrate this diagnostic approach, consider the `Lang-37b` defect. The actual bug is located at line 2962 in `ArrayUtils.java`. When a failing test is executed, it generates the following stack trace:

Listing 3.1: Stack trace generated by the failing test for Lang-37b.

```
at java.base/java.lang.System.arraycopy(Native Method)
at org.apache.commons.lang3.ArrayUtils.addAll(ArrayUtils.java:2962)
at org.apache.commons.lang3.ArrayUtilsAddTest.testJira567(ArrayUtilsAddTest.java:40)
...
at junit.framework.TestCase.runTest(TestCase.java:176)
```

In this scenario, a human developer would immediately scrutinize the code positioned near line 2962 of `ArrayUtils.java` because it appears in the top-most non-native frame. While an SBFL model would assign the same "covered" status to line residing in covered blocks of `addAll` method, our ST Relevance feature quantifies this "closeness to crash" intuition of developers. This provides the DLFL model with a sharp numerical gradient, allowing it to isolate the faulty statement from its benign neighbors. To operationalize this insight, we developed the following structured extraction and mathematical formulation.

3.3.2 Feature Formulation and Extraction Process

The process of calculating the ST Relevance feature, as illustrated in Figure 3.4, follows three distinct stages executed by our automation tool: (1) **Failure Capture**, where full stack traces are recorded for every failing test; (2) **Trace Parsing**, where each frame f is mapped to its function and line number with a positional index; and (3) **Relevance Scoring**, where these components are integrated into a single value based upload proposed decay-based formula.

For each source code line s, the Stack Trace Relevance is formally defined as:

$$STRelevance(s) = \max_{f \in stackTrace(P)} \left(\frac{1}{position(f) + 1} \times e^{-distance(s,f)^2} \right) \tag{3.1}$$

The specific components of this formulation are defined as follows:

- $stackTrace(P)$**:** This represents the aggregate set of all stack frames f extracted from all failing test cases of the faulty program P. It provides a comprehensive map of the failure-inducing call hierarchy.

- $position(f)$**:** This is the zero-based index of frame f within its respective stack trace. A value of 0 denotes the top-most (most proximal) frame where the crash was detected. As the value increases, the frame's relevance decreases, reflecting the depth of the call stack toward the program entry point.

- $distance(s, f)$**:** This denotes the structural distance, measured in lines of code, between the target source line s and the specific line number referenced in frame f. To maintain context integrity, the distance is only calculated if s and f reside within the same method; otherwise, the distance is treated as infinite, resulting in a score of zero.

The formulation utilizes two primary decay mechanisms to reflect the likelihood of a fault:

- **Stack Depth Weighting (Rational Decay):** The term $\frac{1}{position(f)+1}$ prioritizes frames at the top of the stack, effectively modeling the developer's focus on the immediate "crash site" within the source code.
- **Spatial Distance Penalty (Gaussian Decay):** The term $e^{-distance(s,f)^2}$ penalizes lines as their physical distance from the referenced frame increases. This Gaussian decay captures the "neighborhood effect" of a fault, where the exact line in the frame receives the maximum score, while adjacent lines—which may contribute to the state corruption leading to the crash—receive a decayed but non-trivial score.

A critical aspect of this formulation is the use of the **maximum (**max**)** operator. Since a specific code line s may be associated with multiple frames across several failing test cases (e.g., recursive calls or different failing paths), we take the maximum value to capture the line's strongest signal of relevance. By taking the maximum value, we capture the line's **strongest signal of relevance** to any failure point. This ensures that a code element is characterized by its most proximal relationship to a crash, preventing its diagnostic significance from being diluted or averaged out by less relevant appearances in deeper stack frames or distant test scenarios.

3.4 Automated MBFL-based DLFL Dataset Construction Tool

To implement these strategies, we developed a comprehensive automation tool consisting of approximately 6,000 lines of Python code. This infrastructure modularizes mutation management, automates feature extraction, and synthesizes the final dataset.

A key technical contribution of this tool is its integration with an enhanced version of the MUSIC++ [32] (for C/C++ programs) mutation engine to selectively generate mutants based on the optimized target line selection ratio r and mutant count per line m parameters. The tool implements an automatic pipeline that manages program builds, test executions, and the extraction of all 9 features (6 SBFL, 2 MBFL, and 1 ST Relevance). To ensure reliability and ease of use, the tool is well-documented in Appendix A, covering detailed setup procedures, parameter configuration, and troubleshooting protocols. This infrastructure was essential for bridging the gap between theoretical optimization and practical deployment in Company L's C/C++ environments.

3.5 Summary

In this chapter, we presented a systematic methodology to address the cost and accuracy challenges of DLFL. By identifying and optimizing key MBFL parameters (target line selection ratio r, mutant count per line m) and introducing the ST Relevance feature, we establish a robust framework for practical fault localization. Our 6,000-line automation tool serves as the backbone for this methodology, enabling automatic dataset construction from raw source code and its test suite. The following chapter describe the exploratory study conducted to determine the optimal values for target line selection ratio and mutant count per line before applying them to military defense software.

Table 4.1: Technical Specifications and Fault Distribution of the Defects4J Projects utilized for the Exploratory Study.

Identifier	**Project Name**	**# of faults**	**Avg. # Tests**	**LOC**
Chart	JFreeChart	26	2201	96K
Lang	Apache commons-lang	61	2291	22K
Math	Apache commons-math	106	4379	84K
Mockito	Mockito	38	1379	12K
Time	Joda-Time	26	4041	28K

Chapter 4. Exploratory Study Setup

This chapter establishes the experimental framework designed to evaluate the systematic DLFL dataset construction methodology proposed in this thesis. Before applying the localization techniques to complex, high-reliability military defense software, it is essential to identify the optimal parameter configuration that provides a balance between computational efficiency and diagnostic precision in a controlled environment.

The exploratory study serves two primary purposes: (1) to identify the optimal inflection points for the target line selection ratio (r) and mutant count per line (m) to minimize construction costs, and (2) to empirically validate the accuracy gains provided by the novel Stack Trace (ST) Relevance feature. We utilize the Defects4J open-source benchmark to conduct these investigations, supported by a large-scale parallelized infrastructure and rigorous statistical validation.

4.1 Research Questions

The primary objective of this study is to establish a standardized, cost-effective guideline for constructing DLFL datasets. We address the following three research questions (RQs) to evaluate the trade-offs between cost and performance:

- **RQ1 (Target Line Selection Ratio):** What is the impact of varying the selection ratio r (from 10% to 100%) on the overall dataset construction cost and the resulting DLFL model accuracy?
- **RQ2 (Mutant Count Per Line):** How does reducing the mutant density m (from 10 down to 1) influence the computational overhead and the model's ability to precisely localize the faulty code line?
- **RQ3 (ST Relevance Feature Effectiveness):** To what extent does the incorporation of the proposed Stack Trace Relevance feature improve the fault localization precision of the DLFL model compared to traditional features (use of only SBFL and MBFL features)?

4.2 Subject Programs: Defects4J Benchmark

The exploratory study employs **Defects4J (v1.2.0)** [12], the standard benchmark for evaluating automated fault localization. As shown in Table 4.1, Defects4J provides a curated collection of real-world faults extracted from mature open-source Java projects.

For this study, we selected five representative projects: Chart (a visualization library), Lang (Apache Commons utility library), Math (mathematical computation library), Mockito (a mocking framework), and Time (Joda-Time date/time library). These projects collectively encompass **257 real-world faults**, each accompanied by a comprehensive test suite and a confirmed ground-truth fix. Utilizing this benchmark allows for an objective and reproducible assessment of how our optimization parameters and context-aware features generalize across different software structures and logic types.

4.3 DLFL Dataset Construction Infrastructure

To manage the high processing demands of mutation analysis and feature extraction across 257 faults, we utilized our 6,000-line Python dataset construction tool within a parallelized environment.

4.3.1 Distributed Computing Environment

Building MBFL-based datasets is prohibitively expensive on single-node systems. Consequently, we deployed a distributed infrastructure consisting of 30 independent Linux-based machines. Each node is equipped with an AMD Ryzen 7 3800XT 8-core CPU and 32GB of RAM. This parallelized setup enabled the simultaneous execution of mutation generation and test suites, drastically scaling the MBFL pipeline which requires thousands of test runs for each mutant.

4.3.2 Feature Extraction Tools

Feature extraction was managed automatically by our tool, integrating with PITest [5] for mutation analysis in Java. The tool 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.

4.4 Model Training and Evaluation Protocol

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.

To ensure the reliability of our findings and mitigate the risk of overfitting, we employed a **10-fold cross-validation protocol repeated 10 times**. In each iteration, the 257 faults were randomly partitioned into ten folds; the model was trained on nine folds and evaluated on the held-out fold. This 10x10 repetition (100 total iterations) provides statistically robust estimates of fault localization performance, accounting for the inherent variability in neural network training and data partitioning.

4.5 Evaluation Metrics and Statistical Analysis

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

4.5.1 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 RQ1 and RQ2.

4.5.2 Fault Localization Accuracy

Accuracy is evaluated at the **statement level**, aligning with practical debugging workflows where developers inspect code line-by-line. We employ two primary metrics:

- **Top-N:** The number of faults where at least one faulty line 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 statement of a bug. This serves as a direct proxy for the *debugging effort*, as it represents the number of lines a developer must inspect before finding the root cause (lower value is better).

4.5.3 Statistical Significance Testing

To determine whether the performance differences between various parameter configurations are statistically significant, we use the **Mann-Whitney U Test** [1]. This non-parametric test is used to compare the suspiciousness rankings produced by different models (e.g., 70% vs. 100% selection). We apply a significance level of $\alpha = 0.05$; a p-value below this threshold indicates that the observed optimization is statistically sound, ensuring that our cost-reduction guidelines do not compromise diagnostic integrity.

Chapter 5. Exploratory Study Results

This chapter presents the empirical results of the exploratory study conducted on the Defects4J benchmark. The primary objective of this analysis is to discover the optimized parameter configurations that minimize the computational cost of mutation-based dataset construction while preserving diagnostic integrity and verify the effectiveness of the proposed Stack Trace (ST) Relevance feature before its application to military defense software systems.

By interpreting the experimental data, we address the three research questions established in Chapter 4. We specifically focus on finding the optimal parameter configurations for the target line selection ratio (r) and mutant count (m) (parameters identified in Section 3.2) as the primary drivers of the construction bottleneck, while simultaneously validating the accuracy gains provided by the Stack Trace (ST) Relevance feature. The findings presented here provide the empirical foundation and standardized guidelines necessary for scalable Deep Learning-Based Fault Localization (DLFL) deployment.

5.1 RQ1: Impact of Target Line Selection Ratio (r)

As detailed in the MBFL process analysis in Section 3.2, the number of lines selected for mutation (r) directly dictates the scope of the dynamic feature extraction phase. Traditional exhaustive approaches analyze 100% of executed lines, which leads to the prohibitive computation costs. This section evaluates how reducing r through SBFL-based ranking (Ochiai) impacts localization accuracy and construction efficiency.

We varied r from 100% down to 10%. The results, summarized in Table 5.1, show a near-linear correlation between the selection ratio and processing time. However, our goal was to identify the minimum ratio that maintains statistical equivalence to the full baseline. Our analysis confirms that a **70% selection ratio** serves as the optimal threshold.

By limiting mutation to the top 70% of suspicious lines, we reduced the dataset construction time by **29.8%** (from 198.2 to 139.1 CPU-hours). While Top-5 accuracy saw a minor decrease from 48.0% to 45.7%, the Mann-Whitney U test yielded a p-value of **0.0750**. Since $p > 0.05$, we fail to reject the null hypothesis, concluding that the performance loss is not statistically significant. Reducing r below this threshold, however, led to sharp declines in accuracy, indicating that 70% is the inflection point where efficiency is maximized without compromising the diagnostic signals.

Key Finding (RQ1)

Mutation analysis can be restricted to the **top 70% of suspicious lines**, reducing individual extraction costs by **29.8%** while maintaining a diagnostic performance that is **statistically equivalent** ($p = 0.0750$) to the exhaustive baseline.

5.2 RQ2: Impact of Mutant Count Per Line (m)

Following the optimization of the selection scope in RQ1, we investigated the second key cost-driving parameter identified in Section 3.2: the density of mutants generated per line (m). Building on the 70%

Table 5.1: Accuracy and Cost Trade-Off Analysis on Defects4J (RQ1). The table presents the trade-off between the percentage of suspicious lines selected (Line Selection Ratio) and the resulting accuracy and execution time, while keeping the mutant count constant at 10. The row in **bold** (70%) indicates the optimal configuration where cost is reduced significantly without a statistically significant loss in accuracy ($p > 0.05$).

Line Selection Ratio(%)	**Mutants per Line**	**Top-1**	**Top-3**	**Top-5**	**MFR**	**DLFL Process Time (Hours, Δ %)**	**Stat. Test (vs. Baseline)**
		Mean (Δ %)	**Mean (Δ %)**	**Mean (Δ %)**			**p-value**
100%	10	20.5 (0.0)	38.2 (0.0)	48.0 (0.0)	29.9	198.2 (0.0)	Baseline
90%	10	20.1 (-1.9)	37.6 (-1.6)	47.6 (-0.9)	30.7	177.8 (-10.3)	0.5536
80%	10	19.8 (-3.2)	37.5 (-1.9)	47.0 (-2.1)	30.0	158.9 (-19.8)	0.5967
70%	**10**	**19.7 (-3.8)**	**36.0 (-5.9)**	**45.7 (-4.9)**	**31.4**	**139.1 (-29.8)**	**0.0750**
60%	10	19.2 (-6.5)	35.5 (-6.5)	46.0 (-3.2)	33.1	119.2 (-39.9)	0.0495
50%	10	18.4 (-10.1)	34.1 (-10.1)	44.4 (-6.5)	34.4	99.8 (-49.7)	0.0027
40%	10	17.9 (-12.9)	33.9 (-12.9)	43.5 (-10.1)	32.9	74.4 (-62.5)	0.0004
30%	10	16.6 (-19.2)	32.6 (-19.8)	42.4 (-19.7)	36.7	49.8 (-74.9)	0.0000
20%	10	16.5 (-19.7)	32.8 (-19.8)	43.2 (-18.0)	38.6	29.0 (-85.4)	0.0000
10%	10	16.8 (-18.0)	33.7 (-19.2)	43.9 (-19.7)	40.8	11.0 (-94.5)	0.0000

Table 5.2: Accuracy and Cost Trade-Off Analysis on Defects4J (RQ2). The table presents the trade-off between the number of mutants per line and the resulting accuracy and execution time. For this analysis, the Line Selection Ratio is fixed at 70% (the optimal value from RQ1), except for the baseline (100%). The row in **bold** (3 Mutants) indicates the optimal configuration where computational cost is minimized (74.6% reduction) without a statistically significant loss in accuracy ($p > 0.05$).

Line Selection Ratio(%)	**Mutants per Line**	**Top-1**	**Top-3**	**Top-5**	**MFR**	**DLFL Process Time (Hours, Δ %)**	**Stat. Test (vs. Baseline)**
		Mean (Δ %)	**Mean (Δ %)**	**Mean (Δ %)**			**p-value**
100%	10	20.5 (0.0)	38.2 (0.0)	48.0 (0.0)	29.9	198.2 (0.0)	baseline
70%	10	19.7 (-3.8)	36.0 (-5.9)	45.7 (-4.9)	31.4	139.1 (-29.8)	0.0750
70%	9	19.4 (-5.5)	36.4 (-4.8)	46.5 (-3.2)	31.3	129.7 (-34.6)	0.1977
70%	8	19.1 (-6.8)	36.4 (-4.8)	46.2 (-3.7)	31.3	118.6 (-40.2)	0.1533
70%	7	19.0 (-7.4)	36.6 (-4.3)	45.8 (-4.7)	30.5	107.0 (-46.0)	0.1286
70%	6	19.8 (-3.4)	36.3 (-5.1)	46.5 (-3.2)	31.2	94.7 (-52.2)	0.1836
70%	5	19.3 (-5.7)	35.8 (-6.3)	45.3 (-5.7)	31.2	81.5 (-58.9)	0.0753
70%	4	19.7 (-4.0)	36.8 (-3.8)	46.1 (-3.9)	31.3	66.6 (-66.4)	0.1024
70%	**3**	**18.9 (-7.8)**	**36.2 (-5.2)**	**46.0 (-4.2)**	**31.0**	**50.4 (-74.6)**	**0.0792**
70%	2	19.0 (-7.4)	35.9 (-6.1)	45.1 (-6.0)	32.0	34.1 (-82.8)	0.0296
70%	1	17.4 (-15.4)	34.6 (-9.6)	44.0 (-8.3)	32.8	17.5 (-91.2)	0.0015

Table 5.3: Ablation Study of Feature Groups on Defects4J (RQ3). The table shows the Top-N accuracy for different feature combinations. All Δ % values represent the percentage change in accuracy compared to the **SBFL+MBFL** baseline.

Technique	**Top-1**		**Top-3**		**Top-5**	
	Acc.(%)	Δ %	Acc.(%)	Δ %	Acc.(%)	Δ %
SBFL+MBFL	17.7	0.0	33.2	0.0	41.5	0.0
SBFL+MBFL+ST	**18.9**	**6.8**	**36.2**	**9.3**	**46.0**	**11.0**

selection ratio established in the previous section, we systematically reduced m from 10 down to 1.

The experimental results in Table 5.2 demonstrate that construction cost is highly sensitive to mutant density. By restricting the analysis to **3 mutants per line**, the total processing time was drastically reduced to 50.4 hours (representing a **74.6% cumulative reduction)** compared to the exhaustive baseline (100% lines, 10 mutants).

Statistical validation confirms the viability of this reduction. The comparison between the baseline and the 3-mutant configuration resulted in a p-value of **0.0792** ($p > 0.05$), proving that 3 mutants provide sufficient semantic variety for the DLFL model to learn fault patterns. Reducing the count to 1 or 2 mutants resulted in a significant degradation of Top-N metrics. These findings allow us to finalize the cost-efficient guideline: mutation analysis should be targeted at the top 70% of lines with a density of 3 mutants per line.

Key Finding (RQ2)

A density of **3 mutants per line** achieves a cumulative **74.6% reduction** in dataset construction time (from 198.2 to 50.4 hours) while preserving **statistically identical localization effectiveness** ($p = 0.0792$) compared to computationally expensive exhaustive configurations.

5.3 RQ3: Effectiveness of the ST Relevance Feature

The final component of this study validates the core technical contribution proposed in Section 3.3: the Stack Trace (ST) Relevance feature.

Using the optimized configuration ($r = 70\%, m = 3$), we conducted an ablation study to quantify the accuracy gains. As shown in Table 5.3, the inclusion of ST Relevance (9 features, SBFL+MBFL+ST) provided a consistent boost over the traditional model (8 features, SBFL+MBFL).

Specifically, Top-1 accuracy improved by a relative 6.8% (rising from 17.7% to 18.9%), while Top-3 and Top-5 accuracy saw substantial relative gains of 9.3% (33.2% → 36.2%) and 11.0% (41.5% → 46.0%), respectively.

Furthermore, the model exhibited a marked improvement in the Mean First Rank (MFR) metric, which decreased by 14.4% (from 36.2 to 31.0). This reduction indicates that the ST Relevance feature significantly lowers the manual effort required for debugging by ranking faulty elements more precisely within the suspiciousness list, allowing the DLFL model to isolate faulty statements with higher precision.

Key Finding (RQ3)

The proposed ST Relevance feature yields a robust diagnostic boost, improving Top-N accuracy by **up to 11.0%** and reducing manual debugging effort (MFR) by **14.4%**. This accuracy gain is achieved with **negligible computational overhead**, proving the high utility of runtime failure context.

5.4 Summary

The exploratory study on Defects4J successfully discovered the optimal parameter set for efficient DLFL: a **70% target line selection ratio** and **3 mutants per line**. Combined with the novel **ST Relevance feature**, this methodology achieves high-precision (improvement at most 11.0%) fault localization with only a fraction of the traditional computational cost (74.6% reduction).

By validating these strategies in a controlled open-source environment, we have established a robust methodological blueprint. In the following chapter, we transfer these optimized guidelines and our 6,000-line automatic dataset construction tool to the C/C++ middleware systems of Company L to evaluate their performance in a real-world military defense domain.

Table 6.1: Target Subjects (Company L). Subject names are anonymized to protect proprietary information. AB: Artificial Bugs, FTs: Failing Tests, PTs: Passing Tests

Subjects	**Language**	**Size (LoC)**	**Line Coverage**	**Avg. #FTs (on AB)**	**Avg. #PTs (on AB)**
System_A	C	18,854	57.58%	2.52	74.38
System_B	C	4,870	55.78%	1.38	11.58
System_C	C	6,217	66.93%	3.60	7.10
System_D	C++	10,788	60.92%	3.62	3.18
System_E	C++	8,333	68.36%	3.24	31.88
System_F	C++	12,021	45.78%	5.40	20.88

Chapter 6. Application to Military Defense Software: Case Study with Company L

In this chapter, we validate the practical applicability of the systematic DLFL dataset construction methodology proposed in this thesis, through a comprehensive collaborative research project was conducted with Company L, a leading defense electronics company in South Korea. Building upon the optimized parameters ($r = 70\%, m = 3$) and the novel Stack Trace (ST) Relevance feature discovered in the exploratory study, we transition from controlled benchmarks to real-world military defense software.

The objective of this chapter is threefold: (1) to validate the practical applicability of our cost-reduction strategies on C/C++ military defense SW systems, (2) to detail the technology transfer process including the delivery of an integrated automation tool, and (3) to discuss the operational impact of this research on the industrial partner's software quality assurance workflow. This collaboration demonstrates how theoretical optimization can be transformed into a robust, deployed solution for high-reliability software domains.

6.1 Validation on Military Defense Software

To verify that the guidelines derived from Java-based open-source projects generalize to complex defense systems, we applied the optimized DLFL pipeline to six middleware softwrares provided by Company L.

6.1.1 Subject Systems and Experimental Setup

The validation was conducted on three C programs and three C++ programs, totaling approximately 61,000 lines of code. These systems are currently deployed in aerospace, maritime, and guided weapon applications. Due to strict defense security protocols prohibiting access to historical defect logs, we constructed a validation dataset using 300 artificially injected faults (50 per system) generated via the automation tool's mutation engine (as detailed in Section 3.1). The detailed characteristics of these systems are provided in Table 6.1.

Table 6.2: Fault Localization Accuracy on Company L Defense Software (Configuration: 70% Lines, 3 Mutants, with ST Feature)

Metric	**Top-1**	**Top-3**	**Top-5**	**MFR**
Accuracy (%)	62.7%	82.6%	85.0%	18

6.1.2 Performance and Efficiency Analysis

Applying the 70% target line selection ratio and 3-mutant limit alongside the ST Relevance feature yielded high localization results. As summarized in Table 6.2, the optimized DLFL model achieved a **Top-1 accuracy of 62.7%**, **Top-3 accuracy of 82.6%**, and a **Top-5 accuracy of 85.0%**. Furthermore, the Mean First Rank (MFR) was recorded at 18, indicating that a developer would need to inspect fewer than 20 lines of code on average to locate a root cause in a system spanning tens of thousands of lines.

In terms of efficiency, the use of optimized parameters allowed us to circumvent the prohibitive costs typically associated with exhaustive mutation analysis. This approach, theoretically, achieved a cumulative **79.0% reduction (from 9,081 to 1,907 CPU-hours) in dataset construction time** compared to the exhaustive baseline. These results confirm that the strategies identified in the exploratory study (Chapter 5) are not only robust but also highly transferable to C/C++ environments of military defense software, providing high precision at a fraction of the traditional cost.

6.2 Technology Transfer and Delivered Artifacts

The collaboration resulted in the formal handover of three major artifacts, which have been integrated into Company L's internal research and development infrastructure.

6.2.1 MBFL-based DLFL Dataset Construction Automation Tool

The primary vehicle for technology transfer was the **6,000-line Python-based automation tool**. This comprehensive software package automates the entire DLFL lifecycle: from bug synthesis via an enhanced version of MUSIC++ [32] engine to the extraction of 9 dynamic features (SBFL, MBFL, and ST Relevance). To ensure long-term maintainability, the tool includes exhaustive documentation in Appendix A, covering setup, parameter configuration, and troubleshooting protocols.

6.2.2 Comprehensive MBFL-based DLFL Dataset

Utilizing the tool, we synthesized a large-scale fault localization dataset derived directly from Company L's proprietary code. This artifact comprises over **1.3 million rows of feature data**, providing the company with a unique ground truth baseline for training future autonomous diagnostic models.

6.2.3 MBFL-based DLFL Dataset Construction Configuration Guidelines

We delivered a set of empirically validated guidelines that mandate a 70% selection ratio and a 3-mutant limit. By providing these pre-optimized parameters, we eliminated the need for Company L to perform expensive parameter sweep experiments for every software iteration, establishing a "plug-and-play" framework for efficient DLFL construction.

6.3 Collaboration and Technique Adoption

The research collaboration spanned from December 2023 to September 2025, marked by deep technical integration and knowledge exchange.

6.3.1 On-site Research Integration

A critical phase of the project was an **on-site research dispatch** from June 2024 to August 2024. During this period, the proposed dataset construction tool was integrated directly into Company L's secure development environment. This allowed for real-time validation against evolving software baselines and ensured that the tool was compatible with the company's specific build systems and security constraints.

6.3.2 Technical Seminar and Knowledge Transfer

To ensure the sustainability of the provided capabilities, a technical seminar titled *"Dynamic Methods of Fault Localization (SBFL, MBFL, DLFL)"* was conducted for Company L's research division. This knowledge transfer empowered the internal engineering team to independently operate and extend the DLFL infrastructure, transforming it from a static delivery into a continuous research platform for their next-generation weapon systems.

6.4 Summary

In this chapter, we validated the proposed methodology on real-world military software and detailed its successful deployment at Company L. The results demonstrate that the optimized DLFL approach is both **highly accurate (85.0% Top-5)** with **viable computation cost (79.0% reduction, from 9,081 to 1,907 CPU-hours)**. The delivery of the 6,000-line dataset construction tool and optimized guidelines ensures that the diagnostic capabilities established in this thesis provide a lasting impact on the reliability and maintenance efficiency of military defense software.

Chapter 7. Related Works

Fault localization (FL) has undergone a significant evolution over the past two decades, transitioning from simple statistical heuristics to sophisticated machine learning models. This chapter provides a comprehensive review of the existing fault localization literature, organized into four primary domains: traditional and hybrid localization techniques, deep learning architectures for debugging, strategies for mitigating mutation-based computational costs, and the current research gap regarding systematic dataset construction methodologies. By situating this thesis within the broader academic context, we highlight how our proposed optimization strategies and runtime stack trace relevance features address the critical limitations that currently hinder the adoption of deep learning-based diagnostics to military defense SW.

7.1 Traditional and Learn-to-Rank Techniques

Spectrum-Based Fault Localization (SBFL) represents the foundational paradigm of automated debugging. It operates by analyzing program spectra (summaries of code coverage and test outcomes) to assign suspiciousness scores to individual code elements [41]. Early groundbreaking formulas such as Tarantula [11], Ochiai [27], and DStar [40] established that statistical imbalances between passing and failing executions provide a strong signal for defect location. Later, advanced SBFL methods began incorporating genetic programming to automatically evolve more effective formulas (GP13 [47]) or integrated static code complexity metrics to refine coverage-based signals (FLUCCS [37]). Despite their computational efficiency, SBFL techniques are inherently limited by their reliance on execution frequency, which often fails to capture the semantic nuance required for complex software faults.

To overcome the limitations of SBFL, Mutation-Based Fault Localization (MBFL) was introduced to evaluate the behavioral impact of specific code changes. Pioneer techniques such as MUSE [25] and Metallaxis [30] established the intuition that a faulty statement's signature is best captured through test outcome transitions, specifically Failing-to-Passing (F2P) and Passing-to-Failing (P2F). While MBFL offers significantly higher precision, its practical application is often restricted by the massive computational overhead required to generate and execute mutants. This trade-off between precision and efficiency eventually led to the development of *Learn-to-Rank* approaches like PRINCE [16]. These models demonstrated that fusing multiple signals (including SBFL, MBFL, and structural metrics) through supervised learning could outperform any single heuristic, laying the conceptual groundwork for the deep learning models used today.

7.2 Deep Learning-Based Fault Localization (DLFL)

The application of deep learning to fault localization 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. Convolutional Neural Networks (CNNs), such as CNN-FL [51] and DeepRL4FL [18], 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 [22] and GNet4FL [34], 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 [17] and FusionFL [50] have proven that localization 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 [39] and CodeHealer [49] 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 high-performance studies is a focus on model complexity and feature depth, often at the expense of ignoring the underlying efficiency of the data construction process itself.

7.3 Mutation Cost Reduction Strategies

Given the computational burden of MBFL features, a significant body of research has focused on reducing mutation analysis costs while attempting to preserve diagnostic power. These strategies generally target different stages of the mutation pipeline. At the target line selection stage, techniques like HMER [19] and Statement-Oriented Reduction [20] use lightweight SBFL scores or statistical sampling to prioritize which mutants should be executed. Other researchers have explored predictive analysis, where models like Neural-MBFL [7] or Predictive MBFL [44] attempt to guess whether a mutant will be "killed" by a test suite without performing the actual execution, thereby avoiding unnecessary builds.

Further optimizations have targeted the test execution phase itself [28]. IETCR [38] and FTMES [23] reduce overhead by identifying redundant test cases or prioritizing those most likely to fail, while DMES [21] established the theoretical bounds for minimal mutation execution. Additionally, scope reduction via dynamic program slicing, as seen in SMBFL [2], limits mutation to only the code elements residing on the active failure path. While these techniques offer impressive reductions in execution time, they are typically designed for localizing single bugs within a fixed configuration. They do not address the broader challenge of systematically determining the optimal parameter settings (such as selection ratios and mutant counts) for the large-scale dataset construction required to train robust deep learning models.

7.4 The Critical Research Gap: Dataset Construction

Despite the maturity of DLFL architectures and mutation reduction strategies, a significant gap remains in the systematic methodology for constructing the training datasets themselves. Most leading DLFL research treats dataset construction as a secondary task, often resulting in the adoption of arbitrary or exhaustive default settings. For instance, state-of-the-art models like CodeHealer [49] provide extensive details on attention mechanisms and co-teaching strategies but offer little to no rationale for the parameters used to generate their underlying mutation data. This lack of transparency leads to reproducibility challenges and creates a barrier for industrial practitioners who must balance limited computational budgets with high-reliability requirements.

Furthermore, existing studies are predominantly confined to Java-based open-source benchmarks like Defects4J, leaving the generalizability of these techniques to large-scale industrial C/C++ systems largely unverified. Traditional features also remain focused on statistical execution paths, frequently overlooking the rich, context-aware information available in runtime artifacts like stack traces. By failing

to optimize the "what to generate" aspect of data construction and ignoring runtime error context, prior work has left a vacuum that this thesis aims to fill. We address these gaps by conducting the first rigorous investigation into the cost-accuracy trade-offs of construction parameters and by introducing the ST Relevance feature to provide the context that statistical metrics lack.

7.5 Summary

In summary, the field of fault localization has progressed from manual heuristics to automated deep learning pipelines that synthesize diverse program artifacts. However, the prohibitive cost of building MBFL-based datasets and the absence of failure context-aware runtime features remain significant obstacles. By systematically optimizing the construction parameters and incorporating the debugging intuition of developers, this thesis builds upon the foundations of SBFL and MBFL to provide a practical and scalable framework for the application to military defense SW. The methodologies and tools described in this research represent a necessary transition from theoretical accuracy to real-world operational viability.

Chapter 8. Conclusion

The development of Deep Learning-Based Fault location (DLFL) has long been hindered by the prohibitive computational costs of constructing mutation-based datasets and the absence of a standard open-sourced tool for dataset construction. These challenges serve as a barrier to its practical adoption in military defense software, where reliability is paramount but resources for exhaustive analysis are constrained. This thesis has proposed, implemented, and validated a systematic methodology for DLFL dataset construction that effectively resolves the conflict between computational efficiency and fault localization accuracy.

8.1 Summary of Research Contributions

Through a two-phase empirical study involving both open-source benchmarks and military defense systems, this research has achieved three primary technical and practical contributions:

1. **Discovery of the Optimal Parameter for Efficiency:** This study established the first evidence-based configuration for cost-effective dataset construction. By targeting the **top 70%** of suspicious lines and restricting mutant density to **3 mutants per line**, we demonstrated that it is possible to **reduce construction time by 74.6%** (from 198.2 to 50.4 CPU-hours) on the Defects4J benchmark. Crucially, statistical validation via the Mann-Whitney U test confirmed that this reduction maintains **accuracy equivalence** ($p > 0.05$), proving that the optimized parameters preserve the essential semantic signals required for model training.

2. **Innovation of the Stack Trace (ST) Relevance Feature:** This novel metric quantifies the relevance of source code to the runtime failure context (stack trace). Ablation studies revealed that this feature provides a robust **accuracy boost of +6.8% to +11.0%** across standard metrics, confirming that the proposed feature improves the FL accuracy of DLFL.

3. **Validation on Military Defense SW of Company L:** A major practical contribution of this work is the development of a 6,000-line Python-based dataset construction tool that enables systematic DLFL dataset construction in real-world environments. The methodology was rigorously validated through a case study with Company L using this tool. When applied to six C/C++ middleware systems, the technique achieved a **Top-5 accuracy of 85.0%** with a **79.0% theoretical cost reduction**, drastically decreasing the construction time from **9,081 to 1,907 CPU-hours**. These results confirm that the developed tool and optimized methodology are robust, generalizable, and capable of making DLFL scalable for high-stakes defense applications.

8.2 Technical Impact: Collaboration with Company L

The practical viability of this research was cemented through its direct integration into the DLFL system of Company L. The collaboration resulted in several high-value artifacts:

- **MBFL-based DLFL Dataset Construction Tool:** Delivery of a robust **6,000-line Python-based MBFL-based DLFL dasaset construction tool** that manages the entire DLFL process

for C/C++ environments, integrated into the DLFL infrastructure of Company L.

- **Comprehensive MBFL-based DLFL Dataset:** Synthesis of a dataset comprising over **1.3 million rows** of feature data derived from proprietary 6 C/C++ defense software utilized in aerospace, maritime, and guided weapon systems.
- **Knowledge Internalization:** The technology transfer ensured that the partner's internal research team is empowered to independently conduct future model training and extend diagnostic capabilities.

8.3 Future Directions: Hybridizing DLFL with LLMs

While this thesis establishes a rigorous framework for MBFL-based DLFL, the next frontier lies in the synergy between dynamic analysis and semantic modeling. Current Large Language Model (LLM)-based fault localization approaches excel at code comprehension but often ignore the nuances of runtime behavior [13, 14, 43, 45, 35, 10]. Conversely, the DLFL approach presented here captures rich execution patterns but lacks deep linguistic understanding.

Future work should investigate that combine the systematically constructed execution features from this work with the semantic reasoning of an LLM. Integrating these complementary strengths—dynamic signals and static comprehension—holds the potential to create a truly comprehensive diagnostic system that surpasses the limits of either approach in isolation.

8.4 Closing Remarks

This thesis demonstrates that systematic attention to the data construction process of MBFL-based DLFL is a foundational yet often overlooked pillar of Deep Learning-Based Fault Localization. By identifying optimized parameters and introducing the novel Stack Trace Relevance feature, this work transforms DLFL from a computationally prohibitive theory into a practical, viable, and high-performance methodology. The successful deployment at Company L provides compelling evidence that our approach is a significant step toward making advanced AI-driven diagnostics accessible for real-world, military defense software engineering.

Bibliography

[1] Andrea Arcuri and Lionel 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*, ICSE '11, page 1–10, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985795. URL https://doi.org/10.1145/1985793.1985795.

[2] Nazanin Bayati Chaleshtari and Saeed Parsa. Smbfl: slice-based cost reduction of mutation-based fault localization. *Empirical Softw. Engg.*, 25(5):4282–4314, September 2020. ISSN 1382-3256. doi: 10.1007/s10664-020-09845-4. URL https://doi.org/10.1007/s10664-020-09845-4.

[3] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 117–128, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106255. URL https://doi.org/10.1145/3106237.3106255.

[4] Xin Chen, Tian Sun, Dongling Zhuang, Dongjin Yu, He Jiang, Zhide Zhou, and Sicheng Li. Hetfl: Heterogeneous graph-based software fault localization. *IEEE Transactions on Software Engineering*, 50(11):2884–2905, 2024. doi: 10.1109/TSE.2024.3454605.

[5] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 449–452, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2948707. URL https://doi.org/10.1145/2931037.2948707.

[6] Viktor Csuvik, Roland Aszmann, Árpád Beszédes, Ferenc Horváth, and Tibor Gyimóthy. On the stability and applicability of deep learning in fault localization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 546–555, 2024. doi: 10.1109/SANER60148.2024.00062.

[7] Bin Du, Baolong Han, Hengyuan Liu, Zexing Chang, Yong Liu, and Xiang Chen. Neural-mbfl: Improving mutation-based fault localization by neural mutation. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1274–1283, 2024. doi: 10.1109/COMPSAC61105.2024.00168.

[8] Debolina Ghosh and Jagannath Singh. Spectrum-based multi-fault localization using chaotic genetic algorithm. *Information and Software Technology*, 133:106512, 2021. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2021.106512. URL https://www.sciencedirect.com/science/article/pii/S0950584921000021.

[9] Xiaodong Gou, Ao Zhang, Chengguang Wang, Yan Liu, Xue Zhao, and Shunkun Yang. Software fault localization based on network spectrum and graph neural network. *IEEE Transactions on Reliability*, 73(4):1819–1833, 2024. doi: 10.1109/TR.2024.3374410.

[10] Neetha Jambigi, Bartosz Bogacz, Moritz Mueller, Thomas Bach, and Michael Felderer. Fault localization via fine-tuning large language models with mutation generated stack traces. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 417–428, 2025. doi: 10.1109/ICST62969.2025.10988982.

[11] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization for fault localization. 2003. URL https://api.semanticscholar.org/CorpusID:938359.

[12] René Just, Darioush Jalali, and Michael 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*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628055. URL https://doi.org/10.1145/2610384.2628055.

[13] Sungmin Kang, Gabin An, and Shin Yoo. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi: 10.1145/3660771. URL https://doi.org/10.1145/3660771.

[14] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging. *Empirical Softw. Engg.*, 30(2), December 2024. ISSN 1382-3256. doi: 10.1007/s10664-024-10594-x. URL https://doi.org/10.1007/s10664-024-10594-x.

[15] Donguk Kim, Minseok Jeon, Doha Hwang, and Hakjoo Oh. Pafl: Enhancing fault localizers by leveraging project-specific fault patterns. *Proc. ACM Program. Lang.*, 9(OOPSLA1), April 2025. doi: 10.1145/3720526. URL https://doi.org/10.1145/3720526.

[16] Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. Precise learn-to-rank fault localization using dynamic and static features of target programs. *ACM Trans. Softw. Eng. Methodol.*, 28(4), October 2019. ISSN 1049-331X. doi: 10.1145/3345628. URL https://doi.org/10.1145/3345628.

[17] Xia Li, Wei Li, Yuqun Zhang, and Lingming 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*, ISSTA 2019, page 169–180, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3330574. URL https://doi.org/10.1145/3293882.3330574.

[18] Yi Li, Shaohua Wang, and Tien N. Nguyen. Fault localization with code coverage representation learning. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 661–673. IEEE Press, 2021. ISBN 9781450390859. doi: 10.1109/ICSE43902.2021.00067. URL https://doi.org/10.1109/ICSE43902.2021.00067.

[19] Zheng Li, Haifeng Wang, and Yong Liu. Hmer: A hybrid mutation execution reduction approach for mutation-based fault localization. *Journal of Systems and Software*, 168:110661, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2020.110661. URL https://www.sciencedirect.com/science/article/pii/S0164121220301230.

[20] Yong Liu, Zheng Li, Linxin Wang, Zhiwen Hu, and Ruilian Zhao. Statement-oriented mutant reduction strategy for mutation based fault localization. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 126–137, 2017. doi: 10.1109/QRS.2017.23.

[21] Yong Liu, Zheng Li, Ruilian Zhao, and Pei Gong. An optimal mutation execution strategy for cost reduction of mutation-based fault localization. *Inf. Sci.*, 422(C):572–596, January 2018. ISSN 0020-0255. doi: 10.1016/j.ins.2017.09.006. URL https://doi.org/10.1016/j.ins.2017.09.006.

[22] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming 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*, ESEC/FSE 2021, page 664–676, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3468580. URL https://doi.org/10.1145/3468264.3468580.

[23] André Assis Lôbo de Oliveira, Celso Gonçalves Camilo-Junior, Eduardo Noronha de Andrade Freitas, and Auri Marcelo Rizzo Vincenzi. Ftmes: A failed-test-oriented mutant execution strategy for mutation-based fault localization. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 155–165, 2018. doi: 10.1109/ISSRE.2018.00026.

[24] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1169–1180, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510147. URL https://doi.org/10.1145/3510003.3510147.

[25] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162, 2014. doi: 10.1109/ICST.2014.28.

[26] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3), August 2011. ISSN 1049-331X. doi: 10.1145/2000791.2000795. URL https://doi.org/10.1145/2000791.2000795.

[27] Akira OCHIAI. Zoogeographical studies on the soleoid fishes found in japan and its neighhouring regions-ii. *NIPPON SUISAN GAKKAISHI*, 22(9):526–530, 1957. doi: 10.2331/suisan.22.526.

[28] Francis Palma, Tamer Abdou, Ayse Bener, John Maidens, and Stella Liu. An improvement to test case failure prediction in the context of test case prioritization. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'18, page 80–89, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365932. doi: 10.1145/3273934.3273944. URL https://doi.org/10.1145/3273934.3273944.

[29] Yuqing Pan, Xi Xiao, Guangwu Hu, Bin Zhang, Qing Li, and Haitao 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)*, pages 785–792, 2020. doi: 10.1109/TrustCom50675.2020.00107.

[30] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Softw. Test. Verif. Reliab.*, 25(5–7):605–628, August 2015. ISSN 0960-0833. doi: 10.1002/stvr.1509. URL https://doi.org/10.1002/stvr.1509.

[31] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 609–620. IEEE Press, 2017. ISBN 9781538638682. doi: 10.1109/ICSE.2017.62. URL https://doi.org/10.1109/ICSE.2017.62.

[32] Duy Loc Phan, Yunho Kim, and Moonzoo Kim. Music: Mutation analysis tool with high configurability and extensibility. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 40–46, 2018. doi: 10.1109/ICSTW.2018.00026.

[33] Michael Pradel and Koushik Sen. Deepbugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi: 10.1145/3276517. URL https://doi.org/10.1145/3276517.

[34] Jie Qian, Xiaolin Ju, and Xiang Chen. Gnet4fl: effective fault localization via graph convolutional neural network. *Automated Software Engg.*, 30(2), April 2023. ISSN 0928-8910. doi: 10.1007/s10515-023-00383-z. URL https://doi.org/10.1007/s10515-023-00383-z.

[35] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. Soapfl: A standard operating procedure for llm-based method-level fault localization. *IEEE Transactions on Software Engineering*, 51(4):1173–1187, 2025. doi: 10.1109/TSE.2025.3543187.

[36] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu 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*, ESEC/FSE 2018, page 224–234, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236084. URL https://doi.org/10.1145/3236024.3236084.

[37] Jeongju Sohn and Shin Yoo. FLUCCS: Using code and change metrics to improve fault localisation. In *Proceedings of International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 273–283, July 2017.

[38] Haifeng Wang, Bin Du, Jie He, Yong Liu, and Xiang Chen. Ietcr: An information entropy based test case reduction strategy for mutation-based fault localization. *IEEE Access*, 8:124297–124310, 2020. doi: 10.1109/ACCESS.2020.3004145.

[39] Xu Wang, Hongwei Yu, Xiangxin Meng, Hongliang Cao, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. Mtl-transfer: Leveraging multi-task learning and transferred knowledge for improving fault localization and program repair. *ACM Trans. Softw. Eng. Methodol.*, 33(6), June 2024. ISSN 1049-331X. doi: 10.1145/3654441. URL https://doi.org/10.1145/3654441.

[40] W. Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. Software fault localization using dstar (d*). In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 21–30, 2012. doi: 10.1109/SERE.2012.12.

[41] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Softw. Eng.*, 42(8):707–740, August 2016. ISSN 0098-5589. doi: 10.1109/TSE.2016.2521368. URL https://doi.org/10.1109/TSE.2016.2521368.

[42] Shumei Wu, Zheng Li, Yong Liu, Xiang Chen, and Mingyu Li. Gmbfl: Optimizing mutation-based fault localization via graph representation. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 245–257, 2023. doi: 10.1109/ICSME58846.2023.00033.

[43] Chuyang Xu, Zhongxin Liu, Xiaoxue Ren, Gehao Zhang, Ming Liang, and David Lo. Flexfl: Flexible and effective fault localization with open-source large language models. *IEEE Trans. Softw. Eng.*, 51(5):1455–1471, March 2025. ISSN 0098-5589. doi: 10.1109/TSE.2025.3553363. URL https://doi.org/10.1109/TSE.2025.3553363.

[44] Shunqing Xu, Shumei Wu, Zheng Li, Luxi Fan, and Yong Liu. Predictive mutation-based fault localization: Balancing effectiveness and cost. In *2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, pages 577–582, 2024. doi: 10.1109/QRS-C63300.2024.00078.

[45] Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3623342. URL https://doi.org/10.1145/3597503.3623342.

[46] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In Gordon Fraser and Jerffeson Teixeira de Souza, editors, *Search Based Software Engineering*, volume 7515 of *Lecture Notes in Computer Science*, pages 244–258. Springer Berlin Heidelberg, 2012.

[47] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. Human competitiveness of genetic programming in sbfl: Theoretical and empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 26(1):4:1–4:30, July 2017.

[48] Junji Yu, Yan Lei, Huan Xie, Lingfeng Fu, and Chunyan Liu. Context-based cluster fault localization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, page 482–493, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392983. doi: 10.1145/3524610.3527891. URL https://doi.org/10.1145/3524610.3527891.

[49] Lehuan Zhang, Shikai Guo, Yi Guo, Hui Li, Yu Chai, Rong Chen, Xiaochen Li, and He Jiang. Context-based transfer learning for structuring fault localization and program repair automation. *ACM Trans. Softw. Eng. Methodol.*, 34(4), April 2025. ISSN 1049-331X. doi: 10.1145/3705302. URL https://doi.org/10.1145/3705302.

[50] Yanbo Zhang, Yawen Wang, Dongming Zhu, and Wenjing Liu. Fusionfl: A statement-level feature fusion based fault localization approach. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 37–46, 2024. doi: 10.1109/ICST60714.2024.00013.

[51] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan 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)*, pages 445–455, 2019. doi: 10.1109/SANER.2019.8668002.

[52] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Xiaohong Zhang. A study of effectiveness of deep learning in locating real faults. *Information and Software Technology*, 131: 106486, 2021. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2020.106486. URL https://www.sciencedirect.com/science/article/pii/S0950584920302287.

Appendices

Chapter A. 결함위치추정 데이터셋 생성 도구

The instructions provided in this appendix were prepared specifically for engineers at Company L to enable them to independently construct their own MBFL-based DLFL datasets using the tools developed in this study. This documentation is intended as a practical guide for configuring, executing, and troubleshooting the dataset construction process. The datiled technical explanation of the DLFL dataset construction process is found in Chapter 3.

0. 소개

이 프로젝트는 서브젝트에 대하여 결함위치추정 (FL: Fault Localization) 데이터셋 생성하는 도구로 만들어졌다. 서브젝트에 대한 결함위치추정 데이터셋은 다음과 같은 5개의 단계를 거쳐 생성된다:

1. 변이기반 버그 버전 생성
2. 버그 버전 사용 가능 여부 검증
3. 버그 버전 테스팅 (SBFL와 ST 특징 추출)
4. 버그 버전의 변이 테스팅 (MBFL 특징 추출)
5. 최종 결함위치추정 데이터셋 완성

해당 문서는 결함위치추정 데이터셋 생성 도구의 사용법을 설명한다. 해당 문서에서 "./"의 위치는 "cpp_dlfl_feature_extractor" 폴더를 의미한다.

1. 목차

2. 의존 도구

- LLVM 13.0.1
  - 버전: 13.0.1 (이외의 버전 사용 불가)
  - 설치 방법 링크: https://apt.llvm.org/
  - 설치 명령어:

```
$ wget https:/apt.llvm.org/llvm.sh
$ chmod +x llvm.sh
$ sudo ./llvm.sh 13 all
```

- python 3.8
- Python Modules
  - **실행 명령어**:

```
$ pip install -r requirements.txt
```

- postgresql 16.10
- bear 2.3.11
- GNU make 4.1
- cmake 3.10.2
- rsync version 3.1.2 protocol version 31
- diff (GNU diffutils) 3.6
- GNU patch 2.7.6

3. 도구 빌드 과정

3.1 MUSICUP 빌드

MUSICUP은 C/C++ 소스 코드 파일에 대하여 변이 소스 코드 파일을 생성한다. 결함위치추정 데이터셋 생성에서는 MUSICUP을 활용해서 대상 서브젝트의 소스 코드 파일에 대하여 변이 소스 코드 파일을 생성한다. 이렇게 생성된 변이 소스 코드 파일을 대상 서브젝트에 적용하여 테스트 케이스들을 실행하여 1단계인 변이기반 버그 버전 생성과 4단계인 변이기반 데이터셋 추출에서 유용하게 활용된다.

- “MUSICUP“ 도구의 빌드 명령어는 다음과 같다:

```
$ cd ./tools/MUSICUP/
$ make LLVM_BUILD_PATH=/usr/lib/llvm-13 -j20
```

3.2 extractor 빌드

extractor은 C/C++ 소스 코드 파일에서 라인-함수 매핑 정보를 추출해주는 도구이다. 최종 결함위치 추정 데이터셋을 함수 단위로 평가하기에 라인-함수 정보가 필요한 것이다.

- "extractor" 도구의 빌드 명령어는 다음과 같다:

```
$ cd ./tools/extractor/
$ make -j20
```

4. 실행 파일 개요

4.1 결함위치추정 데이터셋 생성 실행 파일

결함위치추정 데이터셋 생성 도구는 1개 실행 파일(./src/main.py)로 작동한다. (참고 사항) 해당 실행 파일은 ./src/ 폴더에 있으며 해당 위치에서 실행할 수 있다.

- 결함위치추정 데이터셋 생성을 위한 실행 파일 목록:
  1. 1단계: 서브젝트의 소스 코드의 변이를 심어 변이 버그 버전을 생성한다.
  2. 2단계: 생성된 변이 버그 버전들의 사용 가능한 버전을 추려 검증한다.
  3. 3단계: 버그 버전에 테스트를 실행하여 SBFL와 ST 특징을 추출한다.
  4. 4단계: 버그 버전에 변이 테스팅을 진행하여 MBFL 특징을 추출한다.
  5. 5단계: 추출된 동적 특징들을 병합하여 결함위치추정 데이터셋의 완성본을 생성한다.
- 실행 파일 main.py 기본 옵션 설명:

```
usage: main.py [--help] [--verbose] [--debug] --experiment-label
    ↪ EXPERIMENT_LABEL [--subject SUBJECT] [--engine-type ENGINE_TYPE]

C++ Deep-Learning-Based FL Feature Extractor

options:
    -v, --verbose          Increase output verbosity
    -d, --debug            Enable debug mode
    -el EXPERIMENT_LABEL, --experiment-label EXPERIMENT_LABEL
                        Label for the experiment (used in logging and output
                           ↪ directories)
    -s SUBJECT, --subject SUBJECT
                  Specify the subject for the experiment
```

5. 실험 기본 configuration 파일 설정 방법

결함위치추정 데이터셋 생성에 초기 설정 파일 4개가 필수로 준비가 필요하다.

- ./.env: 환결변수 설정
- ./.machine_settings: 사용할 서버에 대한 정보
- ./configs/experiment_setup.json: 핵심 파라미터 (target line selection amount, mutants per line) 설정 값
- ../cpp_research_data/subjects/<subject-name>/: 대상 서브젝트에 대한 설정

5.1 실험에 대한 기본 설정 방법

5.1.1 "./.env" 파일 설정

./.env 파일은 실험에 대한 환경 변수 정보를 설정하는 파일이며 다음과 같은 형식으로 설정한다.

```
# .env file for configuration

# POSTGRESQL CONFIGURATION
DB_HOST=db_host
DB_PORT=db_port
DB_USER=db_user
DB_PASSWORD=db_password
DB=db_name

SERVER_HOME=home_directory_path

RESEARCH_DATA=/absolute_path_to_cpp_research_datassd_home
GCOVR_EXEC=/absolute_path_to_gcovr
GCOVR_VERSION=6.0
```

5.1.2 "./.env"의 설정 값 (변수) 설명

- POSTGRESQL DB 설정
  - DB_HOST: 데이터 저장 할 postgresql DB의 호스트 주소
  - DB_PORT: 데이터 저장 할 postgresql DB의 포트 번호
  - DB_USER: 데이터 저장 할 postgresql DB의 사용자 ID
  - DB_PASSWORD: 데이터 저장 할 postgresql DB의 사용자 비밀번호
  - DB: 데이터 저장 할 postgresql DB의 이름
- SERVER_HOME: 사용하고 있는 서버의 홈 디렉토리
- RESEARCH_DATA: 결함위치추정 데이터셋 저장될 절대 경로 (디렉토리 명은, cpp_research_data)
- GCOVR_EXEC: gcovr가 위치하는 디렉토리의 절대 경로
- GCOVR_VERSION: gcovr의 버전

5.1.3 "./.machine_settings" 파일 설정

./.machine_settings 파일은 실험에서 사용할 서버들에 대한 정보를 설정하는 파일이며 다음과 같은 형식으로 설정한다.

```
{
    "server_address": {
        "cores": 128,
        "homedirectory": "/absolute_path_to_homedirectory"
    }
}
```

5.1.4 "./.machine_settings"의 설정 값 (변수) 설명

- "server_address": 서버의 이름 혹은 ip 주소.
- "core": 서버의 홈디렉토리.
- "homedirectory": 서버의 홈 디렉토리.

(참고 사항) 추가적으로 현재 사용중인 서버(main server)로부터 각 분산 시스템 (서버)에 자동 접속을 위해 공개키(public key)가 공유되어 있어야 한다.

5.1.5 "./configs/experiment_setup.json" 파일 설정

./configs/experiment_setup.json 파일은 실험의 핵심 파라미터 (대상 라인 선택 비율, 라인당 변이체 개수) 값을 설정한다.

```
{
    "num_repeats": 10,
    "line_selection_formula": "ochiai",
    "target_lines": [100],
    "mutation_cnt": [10],
}
```

5.1.6 "./configs/experiment_setup.json"의 설정 값 (변수) 설명

- "num_repeats": 실험 (데이터셋 추출) 반복 개수 값.
- "line_selection_formula": 대상 라인 선택 방법 (예: ochiai).
- "target_lines": 대상 라인 선택 비율 (%).
- "mutaiton_cnt": 라인당 변이체 개수.

5.2 서브젝트에 대한 configurations 설정 방법 (예시 기준: libxml2 서브젝트)

서브젝트에 해당되는 모든 정보 (서브젝트 리포지토리, 서브젝트 configurations)는 ../cpp_research_data/subjects/ 디렉토리에 위치 시킨다. 서브젝트에 대한 configuration 설정하는 방법은 5.2.1장 부터 자세하게 설명한다 (설명은 libxml2 서브젝트 기준으로 예를 보인다).

5.2.1 서브젝트의 가장 상위 디렉토리 설정

1. "../cpp_research_data/subjects/" 디렉토리 생성

```
$ cd ../cpp_research_data/
$ mkdir -p ./subjects
```

1. 서브젝트의 리포지토리를 "../cpp_research_data/subjects/libxml2/" 디렉토리 위치에 복사, clone, 혹은 다운로드한다 (서브젝트 리포지토리 디렉토리의 이름은 서브젝트 가장 상위 디렉토리의 이름과 동일하게 내려 받는다).

```
$ cd ./subjects/
$ git clone <libxml2-link> libxml2
```

5.2.2 서브젝트의 실제 버그 (real-world bug) 버전 설정 방법

1. "../cpp_research_data/subjects/libxml2/" 위치에 "real_world_buggy_versions" 이름의 디렉토리를 생성한다. 이 디렉토리는 실제 버그 버전들의 디렉토리로 구성되며 각 버전의 디렉토리가 해당 되는 버전에대한 정보를 담는다.

```
$ cd ../cpp_research_data/subjects/libxml2/
$ mkdir real_world_buggy_versions
$ cd real_world_buggy_versions
$ mkdir HTMLparser.issue318.c
```

- 각 실제 버그 버전들의 세부 정보는 해당 디렉토리에 아래 정보를 담는다:
  - "./real_world_buggy_versions/<버그-버전>/buggy_code_file/<source-file>": 버그 라인을 포함하고 있는 소스 코드 파일.
  - "./real_world_buggy_versions/<버그-버전>/testsuite_info/": 해당 디렉토리에는 다음과 같은 2개의 파일을 포함 시킨다:
    - "failing_tcs.txt": 해당 버그 버전에서 실패하는 *테스트 케이스들의 목록*.
    - "passing_tcs.txt": 해당 버그 버전에서 패스하는 *테스트 케이스들의 목록*.
    - (참고 사항) *테스트 케이스의 목록*은 각 파일의 실행 스크립트의 명칭("<tc-id>.sh")으로 나열한다. 각 실행 스크립트는 하나의 테스트 케이스를 실행하는 명령어가 포함되는 실행 파일이다.

      ```
      TC1.sh
      TC2.sh
      ...
      TC<N>.sh
      ```

  - "./real_world_buggy_versions/<버그-버전>/bug_info.csv": 해당 csv 파일에는 해당 버그 버전에 대해 3개의 feature 정보를 다음과 같은 순서로 담는다:

    ```
    target_code_file,buggy_code_file,buggy_lineno
    libxml2/HTMLparser.c,HTMLparser.issue318.c,3034
    ```

    - "target_code_file": 서브젝트의 리포지토리 디렉토리부터 타깃 소스코드 파일의 상대 경로 (ex. "libxml2/HTMLparser.c")
    - "buggy_code_file": "./real_world_buggy_versions/buggy_code_file/"에 저장한 "<source-file>"의 이름.
    - "buggy_lineno": "<source-file>"에 버기 라인이 위차하고 있는 라인 번호.
      - inference를 위한 데이터 생성시, "buggy_lineno" 값을 "-1"로 입력해준다. 이는 실제 버그의 결함위치를 알지 못하기 때문이다.
- 서브젝트의 실제 버그 버전들은 결함위치추정 데이터셋 추출하는데 있어 필수로 필요한 사항이 아니다.

5.2.3 서브젝트의 configurations 설정

1. "../cpp_research_data/subjects/libxml2/configurations.json" 파일은 서브젝트에 대한 configuration을 설정하는 파일이며 다음과 같은 형식으로 설정.

```
{
    "subject_name": "libxml2",
    "configure_script_working_directory": "libxml2/",
    "build_script_working_directory": "libxml2/",
    "compile_command_path": "libxml2/compile_commands.json",
    "test_case_directory": "libxml2/testcases/",
    "subject_language": "C",
    "target_files": [
        "libxml2/parser.c",
        "libxml2/HTMLparser.c",
        "libxml2/relaxng.c",
        "libxml2/xmlregexp.c",
        "libxml2/xmlschemas.c"
    ],
    "target_preprocessed_files": [
        "libxml2/parser.i",
        "libxml2/HTMLparser.i",
        "libxml2/relaxng.i",
        "libxml2/xmlregexp.i",
        "libxml2/xmlschemas.i"
    ],
    "real_world_buggy_versions": true,
    "environment_setting": {
        "needed": true,
        "variables": {
            "LD_LIBRARY_PATH": "libxml2/.libs"
        }
    },
    "cov_compiled_with_clang": true,
    "gcovr_source_root": "None",
    "gcovr_object_root": "None",
    "test_initialization": {
        "status": false,
        "init_cmd": "None",
        "execution_path": "None"
    }
}
```

- “subject_name“: 서브젝트의 이름.
  - (참고 사항) 5.2.1장의 과정을 수행하여, “configurations.json“ 설정 전에 “./subjects/<subject-name>“ 디렉토리를 먼저 생성해야한다.
- “configuration_script_working_directory“: 서브젝트 configure 명령 실행 스크립트 (“configure_no_cov_script.sh“와 “configure_yes_cov_script.sh“)의 실행 위치를 서브젝트 리포지토리로부터의 상대 경로로 설정한다.
- “build_script_working_directory“: 서브젝트의 빌드 명령 실행 스크립트(“build_script.sh“)의 실행 위치를 서브젝트 리포지토리로부터의 상대 경로로 설정한다.
- “compile_command_path“: 서브젝트 빌드 후 생성 되는 “compile_commands.json“ 파일의 위치를 서브젝트 리포지토리로부터의 상대 경로로 설정한다.
- “test_case_directory“: 해당 서브젝트의 테스트 케이스 실행 스크립트들이 저장된 디렉토리(“test-cases/“)의 경로를 서브젝트 리포지토리로부터의 상대 경로로 설정한다.

  - (참고 사항) 사용자는 필수적으로 “testcases/“ 이름으로 디렉토리를 서브젝트 리포지토리에 생성한 후, 각 테스트 케이스들의 실행 스크립트(“<tc-id>.sh“)를 “TC1.sh“, “TC2.sh“ ... “TC<N>.sh“ 형식으로 만들어 해당 디렉토리의 위치 시켜야 한다.
- “subject_language“: 해당 서브젝트의 프로그래밍 언어를 “”C”“ 혹은 “”CPP”“ 값으로 설정한다.
- “target_files“: 결함위치추정 데이터셋 생성에 타깃으로 하는 소스 코드 파일의 목록 (서브젝트 리포지토리로부터 상대 경로로 표기).
- “target_preprocessed_files“: 결함위치추정 데이터셋 생성에 타깃으로 하는 소스 코드 파일들의 전처리 파일의 목록 (서브젝트 리포지토리로부터 상대 경로로 표기).
- “real_world_buggy_versions“: 실제 버그 버전이 있을 시 “true“, 실제 버그 버전이 없을 시 “false“로 설정한다.
- “environment_settings“: 서브젝트 테스트 케이스를 실행 하기 앞서 요구되는 환경 변수 설정 값.
  - “needed“: 환경 변수 설정 필요 시 “true“, 그렇지 않을 시 “false“로 설정한다.
  - “variables“: 테스트 케이스 실행에 필요한 환경 변수 설정 값의 목록 (“<환경변수>: <환경변수-값>“ 형식으로 설정).
- “cov_compiled_with_clang“: 서브젝트 빌드 명령이 “clang“ 컴파일러를 사용 할 시 “true“, “gcc“ 컴파일러를 사용 할 시 “false“로 설정한다.
- “gcovr_source_root“: “cov_compiled_with_clang“ 변수가 “false“일 때 소스 코드 파일이 위치를 절대 경로로 절대 경로로 설정해주며, “cov_compiled_with_clang“이 “true“일 때는 “”None”“으로 설정한다.
- “gcovr_object_root“: “cov_compiled_with_clang“ 변수가 “false“일 때 소스 코드 파일의 object 파일이 저장되는 위치를 절대 경로로 설정해주며, “cov_compiled_with_clang“이 “true“일 때는 “”None”“으로 설정한다.
- “test_initialization“: 서브젝트의 모든 테스트 케이스들이 공통으로 실행하는 *초기화 코드 라인* 이 있을 시 해당 코드 라인들을 제외하기 위해 필요한 정보를 다음과 같이 설정한다:
  - “status“: 서브젝트의 모든 테스트 케이스들이 공통으로 실행하는 *초기화 코드 라인*이 있을 시 “true“, 그렇지 않을 시 “false“ 값으로 설정한다.
  - “init_cmd“: *초기화 코드 라인*을 추출하기 위해 1개의 테스트 케이스 실행 스크립트의 상대 경로로 설정한다(서브젝트의 리포지토리 경로부터에서의 상대 경로).
  - “execution_path“: “init_cmd“변수의 테스트 케이스 실행 스크립트의 실행 위치를 서브젝트 리포지토리로부터의 상대 경로로 설정한다.

5.2.4 서브젝트의 빌드(build), 정리(clean), configure 명령어 실행 스크립트 파일

1. “../cpp_research_data/subjects/libxml2/build_script.sh“ 이름으로 서브젝트 빌드 명령어를 담은 실행 스크립트 파일로 생성한다.

```
# build_script.sh
bear make -j20 runtest
```

2. “../cpp_research_data/subjects/libxml2/clean_script.sh“ 이름으로 서브젝트 빌드 정리 명령어를 담은 실행 스크립트 파일로 생성한다.

```
# clean_script.sh
make clean
```

3. "../cpp_research_data/subjects/libxml2/configure_yes_cov_script.sh" 이름으로 coverage와 컴파일 DB 추출 설정을 킨 configure 명령어를 담은 실행 스크립트 파일로 생성한다. 이때, configure이 성공할 시 0 값을, 실패할 시 1 값을 반환하게 작성해준다.

```
# configure_yes_cov_script.sh
./make_tc_scripts.py
./autogen.sh CFLAGS='-O0 -fprofile-arcs -ftest-coverage --save-temps' \
    CC='clang-13' \
    CXX_FLAGS='-O0 -fprofile-arcs -ftest-coverage -g --save-temps' \
    CXX='clang++' --with-threads=no
# Check if failed
if [ $? -ne 0 ]; then
    exit 1
fi
```

4. "../cpp_research_data/subjects/libxml2/configure_no_cov_script.sh" 이름으로 coverage와 컴파일 DB 추출 설정을 끈 configure 명령어를 담은 실행 스크립트 파일로 생성한다.

```
# configure_no_cov_script.sh
./make_tc_scripts.py
./autogen.sh CC='clang-13' CFLAGS='-O3' \
    CXX='clang++' CXX_FLAGS='-O0' \
    --with-threads=no
# Check if failed
if [ $? -ne 0 ]; then
    exit 1
fi
```

6. 결함위치추정 데이터셋 생성 단계별 실행 방법

6.1 [1단계] 변이기반 버그 버전 생성

6.1.1 [1단계] 실행 방법

- **실행 명령어**:

```
$ time python3 main.py --experiment-label <experiment-label> --subject <subject-
    ↪ name> --engine-type mutant_bug_generator
```

6.1.2 [1단계]에서 수행되는 작업

1. **변이 생성**:
   - "../cpp_research_data/subjects/<subject-name>/configuration.json" 설정 파일의 "target_files" 변수에 설정된 타깃 파일들의 대해 변이들을 **생성** 한다.
2. **변이 버전 테스트**:
   - "../cpp_research_data/subjects/<subject-name>/configuration.json" 설정 파일의 "test_case_directory" 변수에 설정된 디렉토리 경로에 저장된 테스트 케이스 실행 스크립트들을 각 변이 버전에 실행 한다.

3. **변이 버그 버전 저장**:

    - 각 변이 버전에서 테스트 케이스를 실행하여 1개 이상의 실패하는 테스트 케이스와 1개 이상의 패싱하는 테스트 케이스가 존재하는 경우, 해당 버전 정보는 DB에 기록된다.

6.2 [2단계] 버그 버전 사용 가능 여부 검증

6.2.1 [2단계] 실행 방법

- **실행 명령어**:

```
$ time python3 main.py --experiment-label <experiment-label> --subject <subject>
    ↪  --engine-type usable_bug_selector
```

6.2.2 [2단계]에서 수행되는 작업

1. **변이 버그 버전 선택**:

    - 1단계에서 생성된 변이 버그 버전들의 사용 가능 여부를 확인한다.

2. **실패 테스트 케이스 검증**:

    - 선택된 각 변이 버그 버전 별로 실패하는 테스트 케이스들을 실행하여 커버리즈 정보를 추출한다.

3. **검증된 버그 버전 저장**:

    - 각 변이 버그 버전의 실패하는 테스트 케이스들의 커버리지 정보를 확인하여 사용가능 여부가 검증되면 DB 가능 여부를 참으로 기록한다.
    - **검증 조건**은 다음과 같다:
        - 모든 실패하는 테스트 케이스는 버기 라인을 실행한다.
        - 변이 버그 버전은 1개 이상의 실패하는 테스트 케이스와 1개 이상의 패싱하는 테스트 케이스를 보유한다.

6.3 [3단계] 버그 버전 테스팅 (SBFL와 ST 특징 추출)

6.3.1 [3단계] 실행 방법

- **실행 명령어**:

```
$ time python3 main.py --experiment-label <experiment-label> --subject <subject>
    ↪  --engine-type prerequisite_data_extractor
```

6.3.2 [3단계]에서 수행되는 작업

1. **버그에 대한 기본 정보 (라인, 테스트) 정보 DB에 저장**

2. **테스트 실행 (SBFL와 ST 특징 도출**

    - 각 테스트의 라인 커버리즈 정보를 DB에 저장
    - 실패하는 테스트 케이스 스택 트레이스 정보 DB에 저장

6.4 [4단계] 버그 버전의 변이 테스팅 (MBFL 특징 추출)

6.4.1 [4단계] 실행 방법

- **실행 명령어**:

```
$ time python3 main.py --experiment-label <experiment-label> --subject <subject>
    ↪  --engine-type mutant_mutant_generator
$ time python3 main.py --experiment-label <experiment-label> --subject <subject>
    ↪  --engine-type mutation_testing_result_extractor
```

6.4.2 [4단계]단계에서 수행되는 작업

1. **각 버그에 변이 생성**:
    - 각 버그 버전 별로 "./configs/experiment_setup.json" 설정 파일의 "target_lines" 변수에 설정된 값 만큼의 라인을 선택하여, 각 라인별 "mutation_cnt" 변수에 설정된 값 만큼의 변이를 생성한다.
2. **변이기반 테스팅**:
    - 각 버그 버전에 생성된 변이에 테스트 케이스들을 실행하여 변이기반 테스팅을 수행한다.
3. **변이기반 데이터 DB에 저장**:

6.5 [5단계] 최종 결함위치추정 데이터셋 완성

6.5.1 [5단계] 실행 방법

- **실행 명령어**:

```
$ time python3 main.py --experiment-label <experiment-label> --subject <subject>
    ↪  --engine-type dataset_constructor
```

6.5.2 [5단계]에서 수행되는 작업

1. **최종 결함위치추정 데이터셋 완성**:
    - SBFL, MBFL, 그리고 ST 특징들을 DB 병합하여 최종 결함위치추정 데이터셋을 생성한다. 생성된 데이터셋 (CSV파일을) "../cpp_research_data/<experiment-label>/<subject-name>/" 디렉토리에 저장된다.

Acknowledgments in Korean

먼저 석사과정의 긴 여정 동안 늘 동행해 주신 하나님께 깊은 감사를 드립니다. 연구의 길에서 방향을 잃고 헤맬 때마다 빛이 되어 앞길을 비추시고, 도움이 필요한 순간마다 만남의 축복을 통해 돕는 손길을 허락해 주셨습니다. 부족한 제가 이 모든 과정을 인내하며 무사히 마칠 수 있도록 끝까지 지켜주시고 보호해 주신 은혜에 모든 영광을 돌립니다.

학위 과정 동안 부족함이 많은 저를 인내로 지켜봐 주시고, 학문적으로 한 단계 성장할 수 있도록 세밀하게 지도해 주신 김문주 교수님께 진심 어린 존경과 감사를 올립니다. 교수님께서는 저의 미숙한 부분을 채워주셨을 뿐만 아니라, 연구자로서 스스로 고민하고 성장할 수 있는 기회와 훌륭한 환경을 아낌없이 열어 주셨습니다. 교수님의 가르침이 있었기에 지금의 결실을 맺을 수 있었습니다.

연구실이라는 공간에서 동고동락하며 든든한 버팀목이 되어준 동료들에게도 감사의 마음을 전합니다. 늘 따뜻한 격려와 함께 연구 방향에 대한 아낌없는 조언을 주신 이아청 박사과정, 그리고 늦은 밤까지 연구실을 밝히며 나눈 즐거운 대화로 연구의 고단함을 잊게 해 준 최영석, 강영빈, 장세창 석사과정에게 고마움을 표합니다. 여러분 덕분에 결코 쉽지 않은 과정을 웃으며 완주할 수 있었습니다.

무엇보다 가장 사랑하는 가족들에게 깊이 감사드립니다. 필리핀에 계신 부모님과 미국에 있는 누나, 비록 우리는 멀리 떨어져 지냈지만 보내주신 간절한 기도와 응원 덕분에 외롭고 힘든 시간들을 잘 견뎌낼 수 있었습니다. 지칠 때마다 영상 통화 너머로 전해지던 가족들의 따뜻한 목소리는 저에게 가장 큰 위로이자 힘이었습니다.

포항에서부터 이 낯선 곳까지 인연을 이어오며 서로에게 힘이 되어준 김규림, 백인선, 서현서, 이준명에게도 특별한 고마움을 전합니다. 연구에 몰두하여 마음의 여유를 잃을 때마다 새로운 활력을 불어넣어 주었고, 잠시 학업을 내려놓고 함께 웃을 수 있는 소중한 쉼표가 되어주었습니다. 덕분에 지치지 않고 건강하게 연구실 생활을 이어올 수 있었습니다.

비록 이 지면에 이름을 모두 담지는 못했지만, 저를 위해 끊임없이 기도해 주시고 아낌없는 격려를 보내주신 모든 지인과 친구들에게 진심으로 감사를 드립니다. 여러분이 보내주신 따뜻한 마음 덕분에 무사히 이 과정을 마칠 수 있었습니다. 보내주신 사랑과 기대를 마음 깊이 새기며, 앞으로 세상에 선한 영향력을 끼치는 연구자로 성장하여 보답하겠습니다.

Curriculum Vitae

Name : Heechan Yang

Date of Birth : September 9, 1999

E-mail : heechan.yang96@gmail.com

Educations

2023. 08. – 2026. 02. M.S. Computer Science, KAIST

2018. 02. – 2023. 08. B.S. Computer Science, Handong Global University, HGU

Publications

1. **Heechan Yang**, Ahcheong Lee, Insup Lee, Hyoju Nam, Namhoon Jung, Kyutae Cho, and Moonzoo Kim, How to Create a Dataset to Train a ML Model for Fault Localization: A Case Study on JsonCpp, *Korea Computer Congress (KCC)*, 2024, pp. 331-333.
2. Eujin Ahn*, **Heechan Yang***, Eunkyoung Jee, Shin Hong, A Neural Language Model-based Text Editing Tool for OCR Result of Unstructured Childhood Education Materials, *Korean Software Congress (KSC)*, 2022, pp. 1,444-1,446.

Awards

1. **Best Poster Award**, Impact of Experimental Configurations on the Performance of Deep Learning-Based Fault Localization (DLFL), *Software Disaster Research Center Summer Workshop*, 2025
2. **Best Presentation Award**, Strategies for Efficient Training Dataset Construction in Deep Learning-Based Fault Localization (DLFL), *Software Disaster Research Center Winter Workshop*, 2024

Certifications

1. Teaching English to Speakers of Other Languages, TESOL

Work Experience

2022. 08. – 2022. 12. Software Developer Internship, United Nations (UN), Office of High Commissioner for Human Rights (OHCHR)

Academic Activities

2025. 03. – 2025. 06. Teaching Assistant, CS458 Dynamic Analysis of Software Source Code, KAIST

2021. 03. – 2021. 06. Teaching Assistant, ITP30008 Object-Oriented Design Pattern, HGU

Other Activities

2023. 08. – 2023. 12. Vice-president of School of Computing, KAIST

2021. 12. – 2022. 06. President of Computer Science & Electrical Engineering, HGU

2021. 07. – 2021. 12. Vice-president of Computer Science & Electrical Engineering, HGU