석 사 학 위 논 문
Master's Thesis

시스템 실행 데이터 기반 함수 수준 회귀 퍼징

System Execution Data-Driven Function-Level Regression Fuzzing

2025

최 영 석 (崔 榮 晳 Choi, Youngseok)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

시스템 실행 데이터 기반 함수 수준 회귀 퍼징

2025

최 영 석

한 국 과 학 기 술 원

전산학부

시스템 실행 데이터 기반 함수 수준 회귀 퍼징

최 영 석

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2024년 12월 20일

심사위원장 김 문 주 (인)

심 사 위 원 고 인 영 (인)

심 사 위 원 유 신 (인)

System Execution Data-Driven Function-Level Regression Fuzzing

Youngseok Choi

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 Computer Science

Daejeon, Korea
December 20, 2024

Approved by

Moonzoo Kim
Professor of Computer Science

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 최영석. 시스템 실행 데이터 기반 함수 수준 회귀 퍼징. 전산학부 . 2025년. 30+iv 쪽. 지도교수: 김문주. (영문 논문)
Youngseok Choi. System Execution Data-Driven Function-Level Regression Fuzzing. School of Computing . 2025. 30+iv pages. Advisor: Moonzoo Kim. (Text in English)

초 록

회귀 버그를 찾기 위해 지금까지 프로그램 전체를 테스팅하는 시스템 퍼징을 활용하는 연구가 이루어져 왔으나, 시스템 퍼징은 탐색 공간이 너무 넓고, 회귀 버그가 발생하는 조건을 만족하는 시스템 입력을 퍼징을 통해 합성하기 어렵다는 한계가 있다. 따라서, 본 연구는 함수 수준 퍼징을 통해 회귀 퍼징을 효율적으로 작은 탐색 공간에서 수행하는 기법을 제안한다. 회귀 테스팅을 함수 수준 퍼징을 할 때, 퍼징 대상 함수를 정하는 문제는 시스템 실행에서 구한 함수 커버리지 기반의 동적 함수 연관도와 정적 호출 거리를 기반으로 결정하고, 회귀 버그에 해당하는 유의미한 크래시를 찾는 문제는 시스템 실행의 함수 입력 데이터로 훈련된 모델을 활용하였다. 실험 결과, 10개의 C 프로그램에 대해 적용하여 실제 오류가 발생한 함수를 특정할 때 acc@10 성능이 최신 기법에 비해 40%p 증가하고, 7개의 프로그램에서 제안한 기법이 보고한 오류 중 상위 20% 안에 대상 회귀 오류가 존재함을 보였다.

핵 심 낱 말 함수 수준 퍼징, 회귀 테스팅, 딥러닝 기반 유효성 추정, 함수 입력, 함수 커버리지

Abstract

Previous studies on detecting regression bugs have primarily utilized system fuzzing, which tests the entire program. However, system fuzzing has limitations, including an excessively broad search space and the difficulty of synthesizing system inputs that satisfy the conditions for triggering regression bugs. To address these challenges, this study proposes a method to perform regression fuzzing more efficiently within a smaller search space by leveraging function-level fuzzing.

When conducting regression testing with function-level fuzzing, two key problems arise: selecting the target functions for fuzzing and identifying meaningful crashes related to regression bugs. To solve the first problem, this study employs dynamic function relevance based on function coverage obtained from system execution, combined with static call distance metrics. For the second problem, a model trained on function input data from system execution is used to detect meaningful crashes associated with regression bugs.

Experimental results show that applying the proposed method to 10 C programs improved the acc@10 performance by 40 percentage points compared to state-of-the-art methods in identifying functions associated with actual errors. Additionally, in 7 programs, the proposed method successfully reported regression errors within the top 20% of crashes.

Keywords function-level fuzzing, regression testing, deep-learning based validity estimation, function input, function coverage

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Modern softwares are composed of large size of code, and there is a need to quickly test if new bugs appear after code update. Regression testing is a testing technique to prevent bugs in frequently updated software, usually done by human programmers. There are some works to automate regression testing with greybox fuzzing, which has become a popular testing technique for its high bug detection ability. While existing approaches focus on system fuzzing which explores the entire project codebase, I propose a novel function-level fuzzing apporach for regression testing which tests only the functions that are affected by the recent patch.

1.1 Greybox Fuzzing

Greybox fuzzing is an automated testing technique which applies genetic algorithm to randomly mutate the test input of the target software. It uses coverage metric as fitness score, so that test inputs that cover new branches are favored and gain more probability to be selected as the mutation seed for the next seed generation. By repeating these selection and mutation steps, fuzzing can generate test cases that reach high coverage of the target software and detect vulnerabilities. For example, in OSSFuzz project maintained by Google, greybox fuzzer such as AFL++ and libFuzzer contributed to find 36,000 software bugs over about 1,000 open-source projects [10]. Also, software company Microsoft also tests their software with greybox fuzzing [1].

Although greybox fuzzing is a highly effective technique, automating certain aspects remains challenging—particularly the development of the fuzzing harness. A fuzzing harness, also referred to as a harness, serves as a bridge, translating the random byte sequences generated by the fuzzer into inputs compatible with the target software. Writing an effective harness necessitates a thorough understanding of the target software, as the structure and semantics of the required input are intrinsically tied to its design and behavior.

1.2 Regression Fuzzing

Regression bugs are defects that occur when previously functioning features stop working after certain changes or updates are made to the software. These bugs are introduced inadvertently during the modification of existing code, often due to new code changes that interfere with the existing functionality.

Regression testing is a testing tqchnique used to quickly find regression bugs. Regression testing is usually done manually by humans, but the more frequently code changes, the more labor-intensive regression testing becomes. Several works [8][49] suggested automated regression testing by using greybox fuzzing. These works give more fitness score to seeds that explore the code region close to lines that the latest patch updates [8], or close to frequently and recently changed lines [49]. All existing regression fuzzing studies correspond to system fuzzing, which tests the entire system execution scenario designed to test the entire codebase using the existing system harness, but with a different fuzzing algorithm that prioritizes testing patched lines.

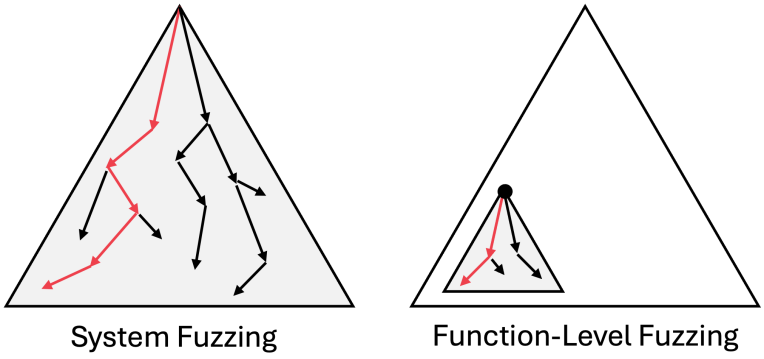

Figure 1.1: The illustration of finding a bug with system fuzzing and function-level fuzzing

1.3 Function-Level Fuzzing

Function-level fuzzing focuses on testing individual functions or methods in isolation from the rest of the software, which is motivated from unit testing. It uses function-level harness which prepares the target function input from raw fuzzing input, and execute the target function. Function-level testing has two advantages for performing regression testing compared to system fuzzing which tests the entire system.

1. **Small search space.** While system fuzzing is trying to find a bug in entire codebase, function-level fuzzing only explores the target function's execution space. If function-level fuzzing is performed on buggy function, it can quickly detect the regression bug located in small search space.

2. **Simpler bug constraint.** Function-level execution always guarantee that the target function is always executed, while system execution may have several constraints to reach the buggy function. The constraint including constraint to reach buggy function to find regression bug can be too complicated to solve by system fuzzing. If function-level fuzzing is performed on buggy function, the reachability condition of buggy condition is always satisifed, so function-level fuzzing can quickly find the regression bug by solving less complicated regression bug constraint.

Figure 1.1 illustrates this idea, where outer triangle is the whole search space of entire project codebase, and grey triangle is search space of each fuzzing approach. Black arrows represent the execution path of passing test case, and the red arrows represesnt the execution path of failing test case, which is a crash. From different size of search space (grey triangle), we can see that finding a bug in function-level search space makes regression bug detection problem much easier compared to system fuzzing. Also, from the length of the execution path of failing test case (red arrows), we can see that function-level fuzzing can detect the regression bug without solving too many branch conditions, while detecting the bug with system fuzzing requires solving many long and complicated conditions. In this way, function-level fuzzing can find regression bugs faster than existing approaches on system fuzzing.

1.4 Thesis Statment and Contribution

There are two challenges to applying function-level fuzzing on regression testing. First, it is important to select the proper functions to perform function-level fuzzing. Function-level fuzzing is only effective when testing the actual buggy function, otherwise it will not detect meaningful regression bugs. Second, it should find regression bugs from the collected function-level crashes. Function-level fuzzing does not

always produce valid test cases because it may prepare invalid contexts as function inputs. To address these challenges, this paper claims,

> **Thesis Statement.** By utilizing system execution data in function-level regression fuzzing, it is possible to select patch-related functions for fuzzing and effectively identify regression bug.

System execution data is the data collected during the execution of system test cases, which is function coverage and function input. Function coverage is the list of functions executed during the entire execution, similar to the well-known coverage metrics such as line coverage and branch coverage. Function input is the set of values that used during each function execution. It includes all values such as function arguments, global variables, and static variables.

Contribution. Based on these two kinds of system execution data, the paper proposes the following contributions:

- **Regression relevance to select patch-related functions to test**. This approach introduces regression relevance metric to find patch-related functions to be tested based on function coverage and static call graph. Compared to state-of-the-art approach [49], the proposed method improves acc@10 by 40%p, which is the success rate that could find buggy functions in top-10 ranking.
- **Function-level test case validity estimator**. It suggests to train a validity estimator model from the function inputs observed in system executions, and use the model to infer the validity of given function-level test case. Using this approach, the regression bug can be found by analyzing only top 20% of function-level crashes having high validity score.
- **Function-level input extraction and interpretation**. It introduces a method to extract function-level inputs from program execution, and interpret the extracted inputs in text and graph to use in various applications. To my knowledge, this is the first work to extract and interpret large number of function-level inputs to train a deep learning model.

1.5 Structure of this paper

I first explain the function-level regression fuzzing approaches in Section 2. It focuses on solving two problems, patch-related function selection and identifying regression bug, by using system execution data. In Section 3, the paper shows the experiment settings and the evaluation results to demonstrate the effectiveness of the proposed approach. In Section 4, I introduce other related works to show how this research is related to existing research topics such as regression fuzzing and automated unit testing. Finally, I conclude my research results and suggest future work in Section 5.

Chapter 2. Regression Function-Level Fuzzing

The overall process of the function-level regression fuzzing is shown in Figure 2.1. As a usual regression testing, the target project codebase and the patch history is given, and the goal is to find a regression bug. First, it selects patch-related functions to perform function-level fuzzing. Second, for each selected function, it creates a fuzzing harness, which loads the function input from fuzzing input, and calls the function. Third, it collects crashes by running function-level fuzzing. Finally, it identifies which crash corresponds to the regression bug among the collected crashes.

In this paper, system execution data is used to improve ① patch-related function selection (Section 2.1) and ② identifying regression bug (Section 2.2). It uses function coverage of each system test case to select functions that are more likely affected by the recent patch(which is ①), and function input data in each function call of system execution to train function a AI-based input validity estimator model (which is ②).

2.1 Patch-Related Function Selection

To perform function-level fuzzing, we need to select functions that need to be tested based on the code changes of the recent commit. In other word, the target project codebase and the code changes are given, and we aim to find the buggy function that the regression bug is located. I will explain why patch-related function selection is a challenging problem with a motivating example (Section 2.1.1), and propose an approach using static call distance (Section 2.1.2) and dynamic function relevance (Section 2.1.3), presenting the final ensembled approach called regression relevance at the end (Section 2.1.4).

2.1.1 Motivating Example

Basic approach will be testing the functions that are updated recently and frequently, as AFLChurn [49] suggested. However, regression bug may appear in the functions that are not updated by recent patches. Figure 2.2 shows the example of regression bug not located in the changed functions. In `main` function, `content` is parsed into message and type (line 35), and these values are passed to `run` function (line 36). `parse` function is the recently updated function, and the patch adds support for message type 2 which was not considered before in `parse` (line 115 - line 118). However, the `run` function does not support for type 2 message, so type 2 message leads to failure by executing the unreachable part written by programmer (line 83). The regression bug exists because the recent patch raised the inconsistency between `parse` and `run` functions for handling type 2 message. Although `parse` function is updated by patch, `run` function has regression bug in this case.

Therefore, regression errors can occur not only in the modified areas, but also in the regions closely related to those areas. To measure how the functions are related to the patch, it utilizes the metrics called static call distance and dynamic function relevance.

2.1.2 Static Call Distance

It can test functions that have a short static call distance to the updated functions. It simply leverages the idea that the more the function is located near the updated functions, the more the function is likely

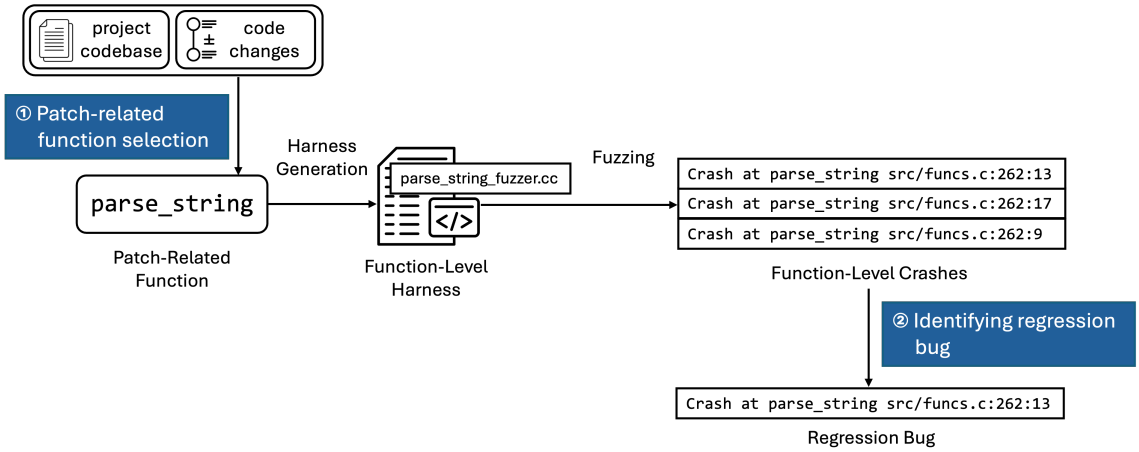

Figure 2.1: The overall process of system execution data-driven function-level regression fuzzing

```
31  function main() {
        // ...
35      message, type = parse(content)
36      run(message, type)
        // ...
41  }
```

```
103 function parse(content) {
104     if (is_type0(content)) {
            // ...
108         return {message: message, type: 0}
109     } else if (is_type1(content)) {
            // ...
113         return {message: message, type: 1}
114     }
115 +   else if (is_type2(content)) {
116 +       message = parse_type2(content)
117 +       return {message: message, type: 2}
118 +   }
        // ...
121 }
```

```
72  function run(message, type) {
        // ...
75      switch type {
76          case 0:
77              run_type0(message)
78              break
79          case 1:
80              run_type1(message)
81              break
82          default:
83              panic("unreachable!")
84      }
85  }
```

Figure 2.2: An example of regression bug which does not located in recently updated function

to be affected by the patch.

Figure 2.3 shows the simple example of static callgraph. Although directed graph is usually used to draw a call graph, I use undirected graph to consider both call and return execution flows. If F_4 is the function updated by the recent patch, test priority of F_1 is higher than F_2 because F_1 is more close to the updated function, F_4. By calculating the call distance to F_4 for all other functions, $F_4 > F_1 = F_5 > F_2 = F_3$ is the test priority of each function.

I define static score which is designed to fall in $[0, 1]$ and to make score proportional to the test priority.

Definition 1 (static score)**.** For each function F, static score is defined as

$$\text{StaticScore}(F) = \frac{1}{1 + \min_{F' \in \text{updated}} \text{CallDistance}(F, F')},$$

where 'updated' is the set of functions changed by the recent patch.

2.1.3 Dynamic Function Relevance

Using only static call distance has limitation because it only considers the static information of the codebase. Therefore, it only provides an approximation of the relationship between functions, and it

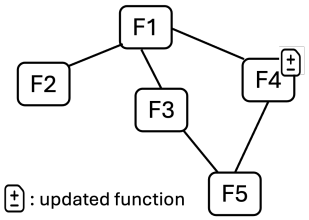

Figure 2.3: An example of static callgraph

	F1	F2	F3	F4	F5
TC1	O	O	X	X	X
TC2	O	X	O	O	X
TC3	O	X	O	O	O
TC4	O	O	O	X	X

: updated function

Figure 2.4: An example of function coverage

may not reflect the actual relationship between functions in system executions.

Dynamic function relevance is the metric for how frequently two functions are executed together in system execution. It was originally proposed to reduce false alarms of unit testing [29, 30]. Among dozens of function relevance/coupling metrics [15, 34, 14, 20, 21], I use dynamic function relevance metric for its intuitive characteristics and its low runtime overhead [32].

Definition 2 (function relevance)**.** Function relevance between two functions F_A and F_B is

$$\text{FunctionRelevance}(F_A, F_B) = P(F_A|F_B) \times P(F_B|F_A),$$

where $P(F_A|F_B)$ is the probability of executing F_A when F_B is executed, and $P(F_B|F_A)$ is the probability of executing F_B when F_A is executed.

From the definition, the function relevance is high when two functions are frequently executed together. Therefore, the function relevance can be used to measure how the given function is affected by the patched functions.

Although the true conditional probabilities are unknown, we can measure them based on the function coverage of each system test case.

$$P(F_A|F_B) = \frac{\#\text{ of system test cases which execute } F_A \text{ and } F_B}{\#\text{ of system test cases which execute } F_B}$$
$$P(F_B|F_A) = \frac{\#\text{ of system test cases which execute } F_A \text{ and } F_B}{\#\text{ of system test cases which execute } F_A}$$

In this way, we can estimate the function relevance based on the function coverage of system test cases.

Definition 3 (dynamic score)**.** Dynamic score for selecting patch-related functions is

$$\text{DynamicScore}(F) = \max_{F' \in \text{updated}} \text{FunctionRelevance}(F, F').$$

where 'updated' is the function updated by the most recent patch. If multiple updated functions exist, the dynamic score is determinted as the maximum function relevance for each target function F and updated function pair.

Algorithm 1 Regression Relevance of Function

Require: Target function f, Changed functions by patch $\{g_i\}_{i=1}^{n}$, System test cases T

$s \leftarrow \min_{g \in \{g_1, \cdots, g_n\}} \textsc{CallDistance}(f, g)$
$s \leftarrow 1/(1+s)$
$d \leftarrow 0$
for $g \in \{g_i\}_{i=1}^{n}$ **do**
    $n_{fg} \leftarrow |\{t \in T : f, g \in \textsc{FunctionCoverage}(t)\}|$
    $n_f \leftarrow |\{t \in T : f \in \textsc{FunctionCoverage}(t)\}|$
    $n_g \leftarrow |\{t \in T : g \in \textsc{FunctionCoverage}(t)\}|$
    $d \leftarrow \max\left(d, \frac{n_{fg}^2}{n_f n_g}\right)$
end for
$r \leftarrow (s+d)/2$

Ensure: Regression Relevance r

For example, suppose function coverage is given as Figure 2.4, where $F_1, F_2, \cdots, F_5$ are functions and TC1, TC2, TC3, TC4 are system test cases. To measure the dynamic score of F_1, it follows following steps.

1. $P(F_1|F_4) = \dfrac{\#\text{ of system test cases which execute } F_1 \text{ and } F_4}{\#\text{ of system test cases which execute } F_4} = \dfrac{|\{\text{TC2, TC3}\}|}{|\{\text{TC2, TC3}\}|} = 100\%$
2. $P(F_4|F_1) = \dfrac{\#\text{ of system test cases which execute } F_1 \text{ and } F_4}{\#\text{ of system test cases which execute } F_1} = \dfrac{|\{\text{TC2, TC3}\}|}{|\{\text{TC1, TC2, TC3, TC4}\}|} = 50\%$
3. $\text{DynamicScore}(F_1) = \text{FunctionRelevance}(F_1, F_4) = P(F_1|F_4) \times P(F_4|F_1) = 50\%$

In this way, the dynamic score of each function can be computed.

$$\text{DynamicScore}(F_1) = P(F_1|F_4) \times P(F_4|F_1) = \frac{|\{\text{TC2, TC3}\}|}{|\{\text{TC2, TC3}\}|} \times \frac{|\{\text{TC2, TC3}\}|}{|\{\text{TC1, TC2, TC3, TC4}\}|} = 50\%$$
$$\text{DynamicScore}(F_2) = P(F_2|F_4) \times P(F_4|F_2) = \frac{|\{\}|}{|\{\text{TC2, TC3}\}|} \times \frac{|\{\}|}{|\{\text{TC1, TC4}\}|} = 0\%$$
$$\text{DynamicScore}(F_3) = P(F_3|F_4) \times P(F_4|F_3) = \frac{|\{\text{TC2, TC3}\}|}{|\{\text{TC2, TC3}\}|} \times \frac{|\{\text{TC2, TC3}\}|}{|\{\text{TC2, TC3, TC4}\}|} = 67\%$$
$$\text{DynamicScore}(F_4) = P(F_4|F_4) \times P(F_4|F_4) = \frac{|\{\text{TC2, TC3}\}|}{|\{\text{TC2, TC3}\}|} \times \frac{|\{\text{TC2, TC3}\}|}{|\{\text{TC2, TC3}\}|} = 100\%$$
$$\text{DynamicScore}(F_5) = P(F_5|F_4) \times P(F_4|F_5) = \frac{|\{\text{TC3}\}|}{|\{\text{TC2, TC3}\}|} \times \frac{|\{\text{TC3}\}|}{|\{\text{TC3}\}|} = 50\%$$

Therefore, the test priority of functions is $F_4 > F_3 > F_1 = F_5 > F_2$ based on the dynamic function relevance score.

2.1.4 Regression Relevance

Although dynamic score measures how the given function is affected by the patched functions, it assumes that the collected system test cases are strong enough to represent all system executions. In other words, if system test cases only execute very small part of the entire codebase, or the distribution of system test cases are skewed to specific code regions, the function relevance-based dynamic score would be inaccurate.

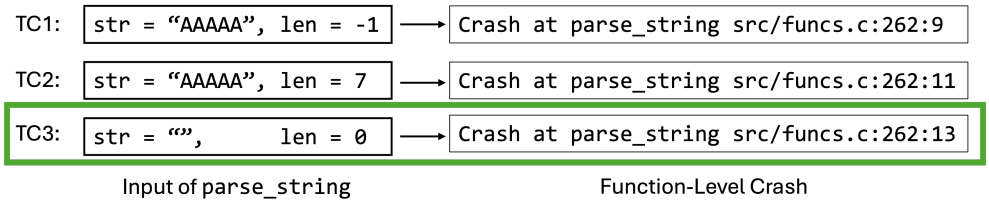

Figure 2.5: An example of valid and invalid function inputs

Static call distance also has limitation since it only considers static information. It does not reflect how functions are related in actual system executions. To overcome these limitations, I use regression relevance which considers both call distance-based static score and function relevance-based dynamic score by averaging both scores.

Definition 4 (regression relevance)**.** Regression relevance of function F is

$$\text{RegressionRelevance}(F) = \frac{\text{StaticScore}(F) + \text{DynamicScore}(F)}{2}$$

Algorithm 1 formally describes how the regression relevance score is computed when the recent patch information is given. First, static score s is measured (line 1-2) based on the minimum static call distance to patched functions. Second, dynamic score d is measured based on the function coverage of each system test case (line 4-8). Finally, the regression relevance r is the average of the static score s the the dynamic score d. In this way, it measures the regression relevance score for each function existing in the project codebase, and we will use top-n functions which have highest regression relevance as the target functions for function-level fuzzing.

2.2 Identifying Regression Bug

Regression function-level fuzzing performs fuzzing on the patch-related functions. It can collect function-level crashes, however some of the function-level test cases produced by function-level fuzzing can be invalid. It is because some functions are designed to assume that the inputs satisfy some constraint, however the given function-level test case may violate the input constraint. This step filters out invalid false positive crashes produced by function-level fuzzing, and it helps identify the regression bug more quickly. I propose a new approach using the function input data observed in system executions to estimate the validity of a given function-level test case.

In this section, I first explain the motivating example of why function-level test cases can be invalid (Section 2.2.1), and then I propose data-driven function input validity estimation model (Section 2.2.2 and Section 2.2.3)

2.2.1 Motivating Example

To understand why function-level test case can be invalid, suppose function-level fuzzing is performed on function `parse_string(char *str, int len)`, and collected three crashing test cases with corresponding function inputs as described in Figure 2.5. Among these crashes, only TC3 is valid because the function `parse_string` in that form usually expects the constraint that length of `str` is identical to value `int` (i.e., `length(str) = len`). Because some functions are designed with some input con-

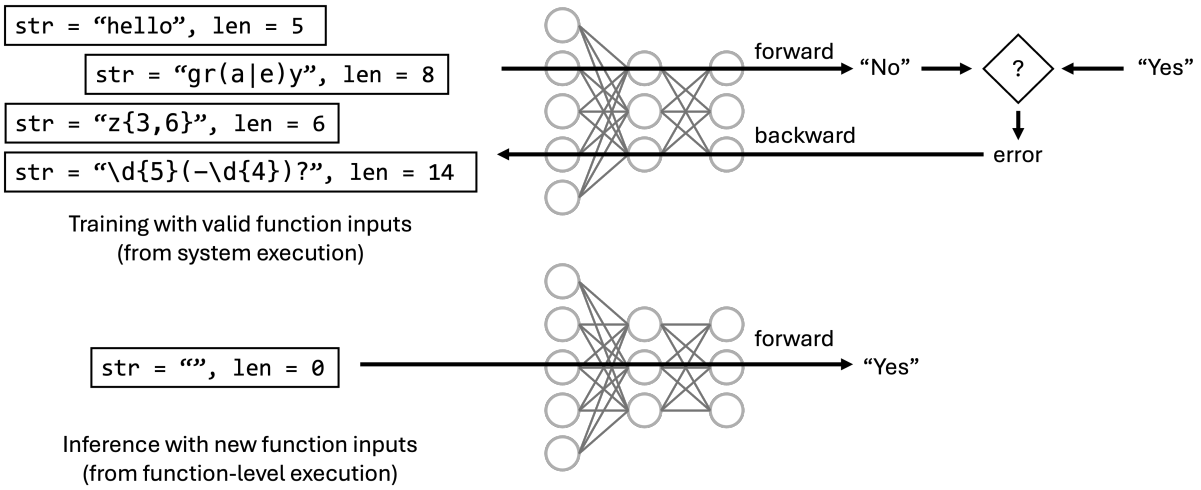

Figure 2.6: Training and inference of the function input validity esimation model

straint assumptions, some function-level crashes violating the input constraint are not useful to find the regression bug.

Therefore, function input validity estimation is required to improve function-level fuzzing. The idea is that the function inputs observed during system execution are always valid. In system execution, the function calls written by a human programmer are expected to work in the correct way or close to the correct way, so that the function call inputs are prepared to satisfy the input constraint of the target function.

For example, one can collect the valid function inputs from system execution, satisfying the input constraint of `parse_string`, and train a function input validity model that learns the expected input constraint. As shown in Figure 2.6, the valid function inputs of `parse_string` observed in system execution satisfy the expected constraint `length(str) = len`. If we train a model that estimates the validity of the given function input based on the patterns of the valid function inputs, then the validity of new function inputs can be measured with this model. Therefore, a deep learning model can measure the validity of function-level inputs generated by function-level fuzzing.

The function input validity model estimates how the given function input I is a valid input of the given function F. Therefore, the inputs are function input F and function F, and it returns the estimated validity score. The problem is how to represent the function input I and function F as a vector, and how to design the model to estimate the validity score. Specifically, the model should consider the following characteristics of the function input data.

- **Highly structured.** Function inputs are usually highly structured, and they can have relations such as pointer-pointee relation and class object-class field relation. The model should be able to interpret the structured function input data.
- **Value-rich.** Function inputs are filled with numeric values such as integers and floats. The model should be able to interpret the concrete numeric values in the function input data.

To consider these characteristics, I propose two approaches to estimate the validity of function-level test cases. The first approach interprets the function input as a text (Section 2.2.2), and the second approach interprets the function input as a graph (Section 2.2.3). I will explain the details of each approach in the following sections.

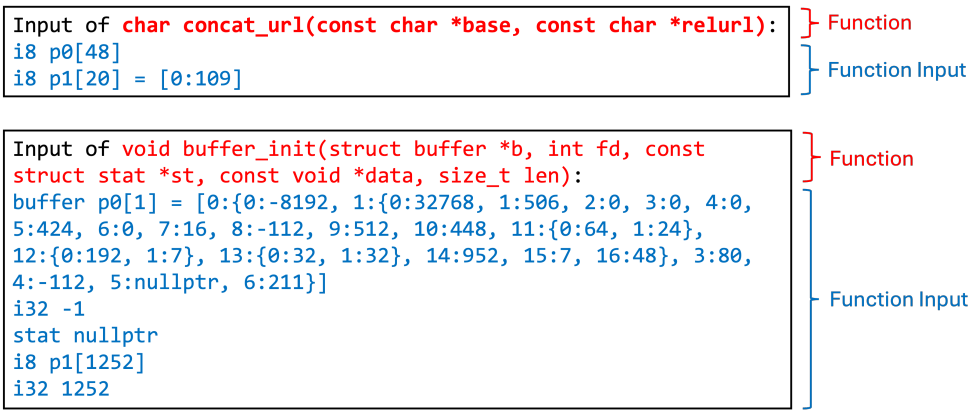

Figure 2.7: The structure of the LLM prompt with function and function input

2.2.2 Function Input Validity Estimation Model: Function Input as a Text

LLMs are powerful for understanding context within sentences and generating coherent text and also have proven to be effective in various natural language tasks. The transformer architecture of LLMs can capture the relations between words and phrases in the text, becuae it uses self-attention mechanism to consider the context of each word in the sentence [45]. Therefore, if we represent the function and function input data as a text, the LLM can learn the relation between the function and the function input data.

To use LLM to estimate the validity of function input, we use pre-trained language model CodeGen [39] which is trained on a large corpus of code snippets. Since the model does not have the information of the function input data, we need to fine-tune the model with the function input data. The model learns the patterns of the valid function inputs observed in system execution, and it can estimate the validity of the given function input based on the learned patterns.

At training phase, the model is fine-tuned with the prompt that includes the function and the function input data. Figure 2.7 shows how the LLM prompt is constructed for training. The prompt begins with "Input of" and the function information follows. Then, in next line, the function input data is written. With this prompt, the LLM learns the patterns of the valid function inputs and how the function input data is related to the function.

At validity estimation phase, only function is given, and the function input part is not given in prompt, and it measures the validity based on the generation probability of the function input data. If the generation probability is high for the given function input, the model estimates the function input as a valid input of the function.

Function Input Representation

Function input has highly structured and value-rich properties, so it is important to make proper representation of the function input as a text. To achieve this, I use only values that the function call accessed in the system execution. If the execution does not make read or write access to vthe value, the value is not considered as the function input. It makes the function input data more concise and focused on the values that are actually used by the function call.

In addition, I use the following rules to convert the function input data into a text.

1. **Numeric value**: We write the numeric value with its type. If the value is integer type, the type is

written as `i16`, `i32`, or `i64` depending on the size of the integer. Similarly, if the value is floating point type, the type is written as `f32` or `f64` depending on the size of the floating point number. For example, the integer value 42 is written as `i32 42`.

2. **Pointer and array**. To represent the pointer and array, we write the type of the pointer or array, the name of the pointer or array, the size of the array, and the inner values of the array. We consider the pointer as an array with size 1. The name of pointer and array is `p0`, `p1`, `p2`, $\cdots$ in the order of appearance in the function input data, and the inner values are written with their index. For example, if the 3rd value of the 32-bit array and the 5th value of the 32-bit array are accessed, and their values are 8 and 0, respectively, it is represented as `i32 p0 = [2:8, 4:0]`.

3. **Struct**. It enumerates each field and value of the struct. For example, if the struct named `foo` contains 32-bit integer 8 and pointer `p0` pointing to 8-bit integer, it is represented as `foo {0: i32, 1: i8 *p0}`.

In this way, the function input data is converted into a text, and it is used as the input of the LLM.

Function Representation

To represent the function as a text, I use the function signature part only because the function signature contains the information of the function name, return type, and parameter types. Although the function body contains the detailed information of the function, I use only function signature because of the maximum length limitation of the LLM input, and the function signature has rich information to represent the function.

Validity Score

The model estimates the validity based on the generation probability of the function input if the function is given. Since the probability vanishes as the length of the input increases, we use normalized generation probability of the function input as the validity score. Specifically, validity score is

$$P(w_1, w_2, \cdots, w_N|\text{decl})^{1/N},$$

where $w_1, w_2, \cdots, w_N$ is the function input data, and decl is the sentence up to the function declaration in the prompt. It is similar to the perplexity of the language model commonly used in language model evaluation [25, 13]

2.2.3 Function Input Validity Estimation Model: Function Input as a Graph

The function input data is highly structured and value-rich, so it is important to make proper representation of the function input data. To achieve this, I propose a graph-based function input validity estimation model. The model interprets the function input data as a graph, where each node represents a value and each edge represents a relation between values. The model embeds the function input data into a vector by using a graph neural network(GNN), and it estimates the validity of the function input based on the learned patterns of the valid function inputs. Similar works have shown that GNNs are effective at learning structured data such as molecular property prediction [42, 46].

The detailed model architecture is described in Figure 2.8. First, function input I and function F are converted into a embedding vector by treating them as a graph and a text, respectively. To achieve

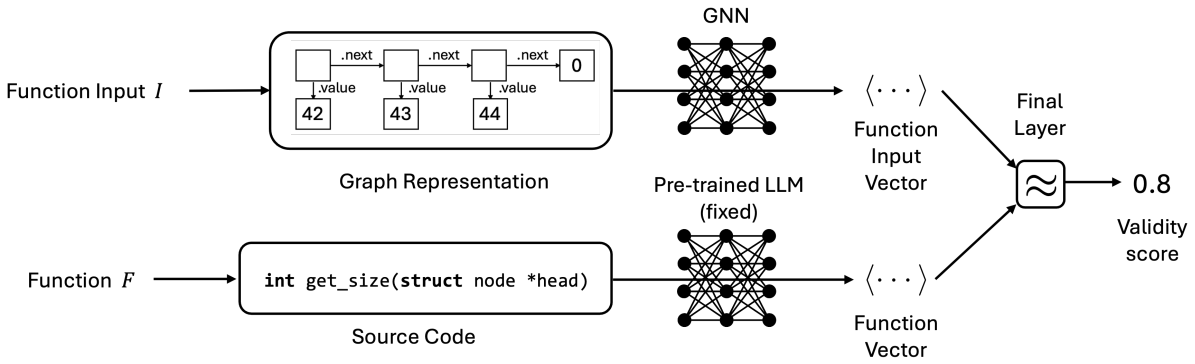

Figure 2.8: The architecture of function input validity estimation model, with function input as a graph

this, a graph neural network embeds the function input I, and a pretrained language model embeds the function F. Second, the final layer aggregates these vectors and returns single real number, which is the validity score. Since the model needs to learn the relation of two inputs from different domain, it uses two-tower architecture commonly used for handling dual-domain tasks such as sentence similarities [38] and recommendation system [48].

In the rest this part, I explain the details of graph-based function input validity estimation model, including function embedding, function input embedding, and final layer.

Function Input Representation

The function input, which includes all values used by given function calls, is usually highly structured and value-rich. Values can have relations such as pointer-pointee relation and class object-class field relation (highly structured), and also be filled with number values such as integers and floating point numbers (value-rich). Both characteristics affect the validity of the function input, so the validity estimation model interprets the function input as a graph, which can effectively process both highly structured and value-rich function input. It encodes the concrete number values into the feature vector of each node, and the value relations into the edge between nodes

Specifically, each node feature vector has dimension 4, where the first element represents the type of the value, where 0 is primitive value, 1 is pointer, and 2 is struct. The remaining three elements are used to express the value itself. If the node represents a numeric value such as integer and float, the second element contains the exact numeric value, and the remaining elements are zero. If node is a pointer, the third element contains the size of the array the pointer points to, and other elements are zero. If node is a struct, the fourth element contains the number of fields in the struct, and other elements are zero.

The edge between nodes is used to express the relation between values. There are two types of relations, which are pointer-pointee relation and struct-field relation. It connects the nodes that have relations, and I do not add any edge features to the edge for simplicity. The edges are undirected to consider information flow in both directions.

The function input data itself is extracted by an LLVM IR pass, which tracks the input values of each function call in the system execution. It records all the values that are used by the function call, and the relations between the values. The function input data is then converted into a graph, where each node represents a value and each edge represents a relation between values. The graph is then fed into the graph neural network to embed the function input data into a vector.

Figure 2.9 shows how the function input is represented as a graph. In this example, the target

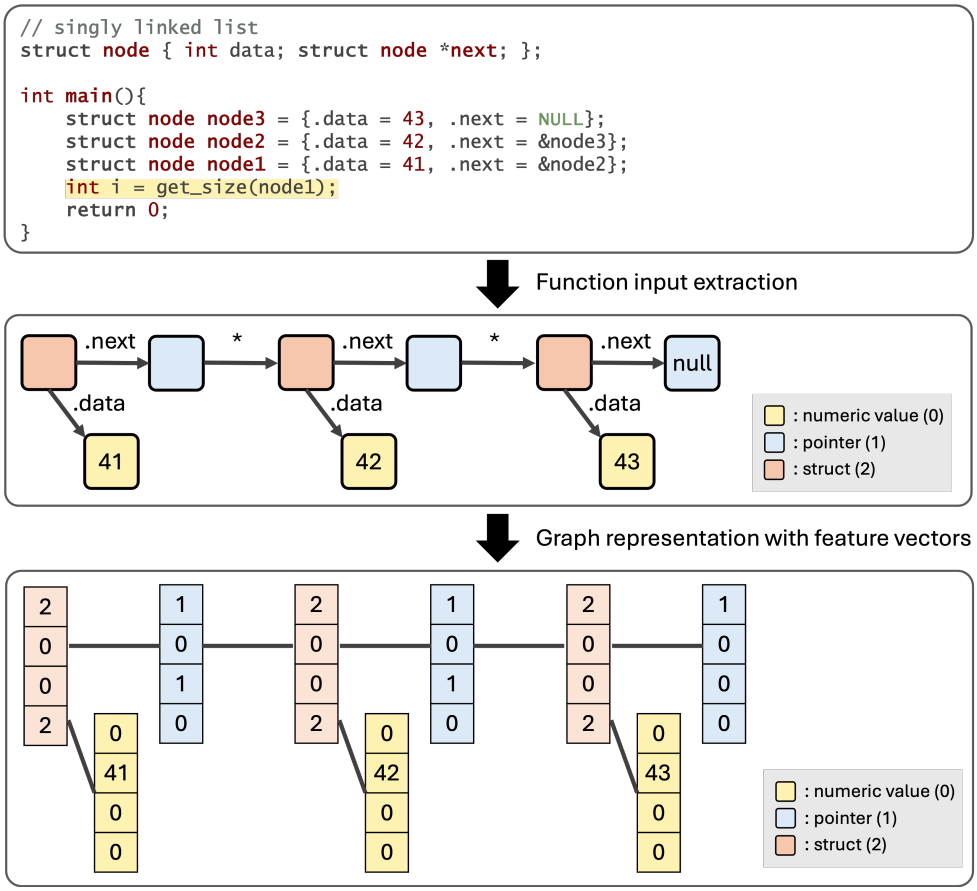

Figure 2.9: Example graph representation of function input

function `get_size` is called with an argument which represents the head node of a singly linked list containing three nodes. 41, 42, and 43 are the integer values in these nodes, and the final node is the null pointer. Function input extraction pass records the values and relations between the values as a graph, which is illustrated in the middle of the figure. The figure at the bottom shows how the node features are represented. For example, the head node has two struct fields, so it is represented as a node with feature vector $\langle 2, 0, 0, 2 \rangle$. The yellow node connected to the head node corresponds to the integer value 41, so it is represented with a node with feature vector $\langle 0, 41, 0, 0 \rangle$. The blue node connected to the head node corresponds to the pointer which points to single linked list node (length 1), so it is represented as the node with feature vector $\langle 1, 0, 1, 0 \rangle$. The rightmost blue node represents null pointer, so it is considered as a pointer node with length 0, so it has feature vector $\langle 1, 0, 0, 0 \rangle$.

Finally, average pooling is used to aggregate the node feature vectors into a single function input embedding vector. The average pooling is used to make the function input embedding vector invariant to the number of nodes in the graph, and it is commonly used for graph neural networks to aggregate the node feature vectors.

Function Representation

To embed function, it uses a pre-trained language model. Now it is natural to treat the programs (or program objects such as functions) as text written in programming language, as number works show that code generational language models are effective at tasks that require understanding on programs

including code completion and summarization [5, 39, 43, 12]. Therefore, we can use a pre-trained language model to effectively embed a function F into a vector.

We use hidden state of the last token of the function F as the function embedding vector, similar to [47]. The hidden state of the last token is considered as the feature vector of the entire function, and it is used as the function embedding vector. To not exceed the maximum token length of the language model, we only use the declaration part of the function for embedding. The declaration part is the first part of the function, which includes the function name, return type, and input arguments. The declaration part is usually short and concise, so it is enough to represent the function's high-level characteristics.

Also, the pre-trained language model is fixed and not fine-tuned during the training of the function input validity model. This is because the language model has already been trained with a large codebase and is expected to have enough knowledge to understand the high-level features of the function.

Validity Score

The final layer of the function input validity model aggregates the function embedding vector and the function input embedding vector, and returns the validity score. I use simple cosine similarity as the aggregation function, which is commonly used for dual-domain tasks. More complicated aggregation function such as multi-layer perceptron or attention mechanism can be used, however I use cosine similarity for simplicity and efficiency.

Therefore, the only trainable parameters are only the weights of the graph neural network, since the pre-trained language model is fixed and final layer is simple cosine similarity. It can be understood that the function input validity model trains graph embedding network to learn only function-specific patterns, and it can be used to estimate the validity of the function-level test cases generated by function-level fuzzing.

2.3 Summary

In this chapter, I proposed a function-level regression fuzzing approach that selects patch-related functions and identifies regression bugs. For patch-related function selection, I proposed regression relevance that considers both static call distance and dynamic function relevance. To measure dynamic function relevance, I use function coverage of system test cases. For regression bug identification, I proposed a function input validity model that estimates the validity of function-level test cases generated by function-level fuzzing. It uses valid function input data observed in system execution to train the model that satisfies the input constraint of the target function. In the next chapter, I will evaluate the proposed approach with real-world regression bugs in open source project, and compare the performance with existing approaches.

Chapter 3. Evaluation

My main hypothesis is that utilizing system execution data in function-level regression fuzzing can help select patch-related functions for fuzzing and effectively identify regression bugs. In this experiment, two research questions were formulated to evaluate the function-level regression fuzzing technique proposed in this study:

- **RQ1**: How does the performance of the regression information-based function selection technique proposed in this study compare to state-of-the-art methods?
- **RQ2**: Can the function input validity estimation model effectively reduce false positives in function-level fuzzing? Specifically, when measuring the validity of function inputs across all function-level crashes, do the true positive function-level crashes rank higher in validity?

To answer RQ1, I compared the proposed regression relevance metric with AFLChurn [49], a state-of-the-art regression fuzzing technique. To answer RQ2, I evaluated the function input validity estimation models by ranking the validity of function-level crashes associated with regression bugs. The following sections describe the evaluation settings and results for each research question.

3.1 Evaluation Settings

3.1.1 Regression Bug Benchmark

I conducted experiments on regression errors from 10 C programs, registered in OSSFuzz, which were used in the regression error benchmark BugOSS [27] and the regression fuzzing study AFLChurn [49]. Only C programs were selected because the fuzzing harness generation engine and function input extraction pass used in this study only support C programs. Table 3.1 lists the target programs and their corresponding regression errors. The table includes the project name, OSSFuzz ID, version number, and the size of the program in lines of code (LoC). The version number indicates the commit hash of the version that introduced the regression error. In this study, the version of bug-inducing-commit was used as the regression testing target.

3.1.2 Patch-Related Function Selection

I compared the function selection method proposed in this study with AFLChurn [49], a state-of-the-art regression fuzzing technique designed to efficiently identify regression bugs. AFLChurn modifies the fuzzing algorithm to prioritize inputs that execute code regions with a history of frequent and recent changes, based on patch history data. To be specific, AFLChurn defines the score for each line as $\frac{\log(\text{churn})}{\text{age}}$, where churn is total number of changes, and age is the number of days since the last change. The score is calculated for each line of code, and I use the maximum score of the lines that the function contains as the AFLChurn score of the function.

I evaluated the effectiveness of each approach by ranking functions associated with regression errors within the top 10 positions. The regression relevance metric proposed in this study was used to rank the functions, and the AFLChurn score was used to rank the functions for AFLChurn. The accuracy of each

Table 3.1: Target regression bugs

Project	OSSFuzz	Version	Size (LoC)	Project	OSSFuzz	Version	Size (LoC)
curl	8000	`dd7521`	110619	libxml2	17737	`1fbcf4`	202998
file	30222	`6de368`	15036	ndpi	49057	`2edfae`	47302
leptonica	25212	`8fc490`	185268	picotls	13837	`7122ea`	18928
libgit2	11382	`7fafec`	159300	readstat	13262	`1de4f3`	25612
libhtp	17918	`3c6555`	14697	yara	38952	`5cc28d`	44531

Table 3.2: Number of collected passing system test cases

Subject	# of System Test Cases	Line Coverage	Branch Coverage
curl-8000	58004	53.90%	46.97%
file-30222	34677	21.18%	18.91%
leptonica-25212	6641	2.69%	4.62%
libgit2-11382	17064	3.38%	2.67%
libhtp-17918	78338	65.21%	60.06%
libxml2-17737	281767	19.13%	21.36%
ndpi-49057	126552	61.09%	57.62%
picotls-13837	2963	4.52%	2.75%
readstat-13262	11660	49.84%	49.91%
yara-38952	16459	19.28%	15.47%
Average	63412.5	30.02%	28.03%

approach was measured by the number of functions associated with regression errors that were ranked within the top 10 positions.

3.1.3 System Execution Data

Patch-related function selection requires function coverage to measure the regression relevance, and function input validity estimator requires function inputs to train the model. This data is collected by running passing system test cases, which were collected by running AFL++ [22] on the pre-regression versions of the software for 12 hours per run, repeated five times to resolve the randomness and collect various function input data. The default setting of OSSFuzz including seed and harness was applied for this process. Table 3.2 shows the number of system test cases for each past regression bug subject.

Function coverage and function input are collected by LLVM pass fully-automatically. Function input includes all the values in the memory accessed by the function call including function arguments, global variables, and static variables. The overhead of the function coverage pass is negligible, but the function input extraction pass makes program take average 15.8 times longer time compared to the default system execution.

3.1.4 Function-Level Fuzzing

Fuzzing harness for each function were automatically generated using the harness generation feature of the concolic testing engine, CROWN 2.0 [26]. These harnesses declare the symbolic variables required for function execution and assign values derived from fuzzing inputs. For functions with pointer-type inputs, I modify CROWN 2.0 to create drivers capable of identifying various memory-related errors. During driver compilation, AFL++'s LAF-Intel [2] was employed to facilitate the generation of more diverse test inputs.

Function-level fuzzing was conducted for the top 10 functions with the highest regression relevance scores for each project, with each harness running for one hour. All experiments were performed on machines equipped with an AMD Ryzen 9 5950X CPU and 32GB of memory under identical conditions. To minimize randomness in experimental results, each experiment was repeated five times, and the average results were reported as the RQ2 result.

3.1.5 Function Input Validity Estimation Model

Function input validity estimation model uses function input data collected from system executions to estimate the validity of function-level test cases. The training data consists top-10 functions with the highest regression relevance scores for each project, and 1,000 function inputs for each function. All experiments were conducted on a single 24GB NVIDIA GeForce RTX-4090 GPU. The detailed settings of each model are as follows:

Function Input as a Text

350M parameter version of CodeGen-multi, a large-scale code generation language model [39] was used. Since CodeGen-multi is trained on diverse programming languages including C/C++, it is suitable to this task as the target subjects are written in C. To reduce the memory required for training, I use LoRA and NF4 quantization techniques [23, 16] for fine-tuning the LLM.

Function Input as a Graph

The function input validity estimation model has GNN and LLM to encode function input and function, respectively. GNN is a trainable network, which uses GCN [31] with 3 layers having 1024 hidden dimension.

The model uses text feature of function, so pre-trained language model Codegen-350M-multi was used [39]. I only used function declaration part to not exceed maximum token length limit. The hidden vector of final token was used as function embedding. In this case, the pre-trained LLM is fixed and the weights are not updated in the training phase.

The final layer produces the validity score from function input vector and function embedding vector. Cosine similarity was used as final layer. Therefore, GNN is the only part that trained in the entire archtiecture. This design makes GNN to focus on learning function input patterns from given raw function inputs, similar to representation learning.

Table 3.3: Regression relevance ranking of functions with regression bug

Subject	# of functions	AFLChurn	Proposed
curl-8000	1008	22	**2**
file-30222	414	**1**	**1**
leptonica-25212	2883	213	**73**
libgit2-11382	3541	**4**	**4**
libhtp-17918	428	19	**1**
libxml2-17737	2793	4	**1**
ndpi-49057	1375	14	**1**
picotls-13837	481	57	**6**
readstat-13262	196	144	**65**
yara-38952	690	6	**1**
Average	1380.9	48.4	**15.5**

3.2 Result

3.2.1 RQ1. Patch-Related Function Selection Performance

The experimental results are summarized in Table 3.3. The proposed method successfully identified target functions within the top 10 rankings for 8 out of 10 programs, achieving an acc@10 of 80%. In contrast, the function selection approach of AFLChurn yielded an acc@10 of 40%, demonstrating that the proposed method improved acc@10 performance by 40 percentage points. This improvement highlights the contribution of the regression relevance metric, which combines static distance and dynamic function correlation, in detecting functions associated with regression errors.

AFLChurn, relying solely on change history, exhibits limitations in identifying functions related to regression errors compared to the proposed method, which analyzes functions associated with modified regions more comprehensively.

For leptonica-25212, the proposed method's regression relevance metric was less accurate. In this case, the system test cases collected from fuzzing does not covered the execution flow that executes both crash-related functions and changed lines, so dynamic function relevances of crash-related functions remain zero. This limitation reflects a structural drawback of dynamic relevance metric used in regression relevance. When system test cases are not powerful enough to see various executions that reaches the changed lines of code, dynamic relevance has low accuracy to find suspicious functions to test.

Summary RQ1: The proposed appraoch improved the accuracy of patch-related function selection by 40 percentage points compared to AFLChurn. The regression relevance metric, which combines static distance and dynamic function correlation, effectively identifies buggy functions associated with regression errors.

Table 3.4: Validity rankings of regression function-level crashes evaluated by the validity estimation models

Two models were evaluated for each project, the LLM based model that uses function input as text (left), and the GNN based model that uses function input as graph (right).

Subject	# of Unique Crashes	Rank (Text)	Rank (Graph)
curl-8000	44.2	11.8	**8.4**
file-30222	25.4	20.4	**2.2**
libgit2-11382	23.8	**4.2**	6.8
libhtp-17918	102.8	**6.8**	10.6
libxml2-17737	74.6	24.0	**8.0**
picotls-13837	76.0	**6.6**	12.4
yara-38952	57.8	**15.4**	32.4
Average	59.4	11.3	11.5

3.2.2 RQ2. Validity Estimation Model Performance for Identifying Regression Bugs

I investigated the extent to which a validity estimation model trained on function inputs extracted from system-level executions can contribute to distinguishing valid and invalid function-level test cases. In order to evaluate the validity estimation model, I collected function-level crashes from fuzzing the top-10 functions with the highest regression relevance scores for each project. The function-level crashes are deduplicated based on crash location, and the ranking of each unique crash is estimated by the ranking of maximum validity among the crashes that have the corresponding crash location. The true positive crashes indicating regression bugs were identified by comparing the crash location with the location of the regression error, and ramining crashes were considered as false positives.

Table 3.4 summarizes the number of crashes collected by fuzzing the top-10 functions with the highest regression relevance and the validity rankings of true positive crashes which represent the regression bug. Among the 10 target programs, leptonica-25212 and readstat-13262 were excluded as they failed to identify functions associated with regression errors within the top 10 functions in patch-related function selection phase. Additionally, ndpi-49057 was excluded from ranking as fuzzing did not yield any true positive function-level crashes because of the limitation of harness generation technique.

As a result, over remaining 7 projects, an average of 59.4 function-level crashes were discovered across the seven programs with regression errors. With the validity estimation model, the average validity ranking of the true positive crashes was 11.3 for the text-based model and 11.5 for the graph-based model. For the text-based model, the average validity ranking of the true positive crashes was 11.3, and for the graph-based model, it was 11.5. The results indicate that the validity estimation model effectively reduces false positives in function-level fuzzing, as the true positive crashes were ranked higher in validity compared to the false positive crashes. This indicates that analyzing only the top 20% of function-level crashes with high validity is sufficient to identify function-level test cases that manifest errors at the same points as system-level regression errors.

The difference between the text-based and graph-based models was not significant. The graph-based model showed slightly better performance in 4 out of 7 projects, but the text-based model showed better

performance in 3 out of 7 projects. Also, the average ranking difference between the two models was only 0.2, which is negligible. This result indicates that both models are effective in estimating the validity of function-level test cases. However, since fine-tuning LLM requires much more time and resources compared to light-weight GNN, the graph-based model is more efficient in practice.

Summary RQ2: The function input validity estimation model enables the identification of regression bugs by ranking the true positive function-level crashes higher in validity. By analyzing only the top 20% of function-level crashes with high validity, it is possible to effectively identify function-level test cases that manifest errors at the same points as system-level regression errors.

3.3 Threats to Validity

A threat to internal validity is possible bugs in function input extraction pass. To control this threat, I tested our implementation extensively to ensure that the function input extraction pass works correctly.

One threat to external validity is the representativeness of the benchmark. I expect that this threat is limited because the benchmark used in this study is a subset of the benchmark used in the previous regression bug study, BugOSS [27] and AFLChurn [49]. I tried to include all the subjects that are used in the previous studies, but some subjects were excluded because they are written in languages other than C or the harness generation failed.

Another threat to external validity is the generalizability of the proposed validity estimation model. The model was trained on a small number of function input data of top-10 functions collected from system-level executions. The model may not generalize well to new function inputs that are not included in the training data and overfit to the current subjects. I did not explore the generalizability of the model because of expensive function input extraction cost, but it is an important future work to investigate the generalizability of the model.

Chapter 4. Related Works

Function level regression fuzzing is an approach to use function-level fuzzing to achieve regression testing. This chapter introduces how function-level fuzzing is related to existing works on regression testing, so that we can understand the difference and novelty of the proposed approach. Also, this chapter covers how function-level fuzzing is related to existing works on automated unit-level testing. Finally, this chapter introduces how the function input validity estimation problem is related to works in neural network testing.

4.1 Regression Testing

Patch modification have long been known to be a main cause of bugs [3, 49], which sparked many researches on regression testing. Directed fuzzing [8, 9, 11, 17, 28] guides the system fuzzer to the target program locations by utilizing control flow distance metric, and regression testing can be achieved by directed fuzzing on the updated program locations by the recent patch.

Some works suggest to use entire commit history to guide the regression testing, not only the most recent patch [49, 33]. AFLChurn [49] is an appraoch to use greybox fuzzing on regression testing which prioritizes the inputs that executes the program locations close to frequently and recently updated lines in the commit history. SyzRisk [33] is a kernel regression fuzzer that uses system fuzzing to find the regression bugs in the Linux kernel. It introduces code change patterns that allow for identifying risky code changes, so that the fuzzer can focus on more important code changes. My approach is limited to the most recent patch, and it can be extended to use the entire commit history to guide the regression testing by updating the patch-related function selection strategy.

Also, directed symbolic execution [6, 7, 37, 44, 36, 41] leverages symbolic execution to guide the search to the target program locations updated by the recent patch. However, directed symbolic execution is known to be slow and expensive, due to the heavy-weight program analysis and constraint solving.

All these works are based on guiding system executions to reach the target program locations updated by recent patches, but they are not focusing on the function-level fuzzing. In contrast, this paper focuses on function-level fuzzing to achieve regression testing, which is more efficient than system fuzzing in terms of small search space and simpler bug constraint.

4.2 Automated Unit-level Testing

Fuzzing with auto-generated harness can be seen as an extension of automated unit testing because it tests individual functions with automatically generated inputs. Automated unit test generation [18, 40] is a technique to generate unit tests automatically, and it is widely used in the software testing community. EvoSuite [18] is a search-based unit test generation tool that uses genetic algorithms to generate unit tests for Java programs. Randoop [40] is a random test generation tool that generates unit tests by executing the target program with random inputs. These tools generate unit tests that cover the target program as much as possible, but they do not consider the validity of the function inputs.

Greybox fuzzing has been widely studied in the software testing community, and several works tried to generate harness automatically to test the target program [26, 4, 24, 19, 35]. These works focus

on library fuzzing or API fuzzing, which is similar to function-level fuzzing introduced in this paper. FUDGE [4] and FuzzGen [24] synthesize fuzz drivers based on existing library consumer projects to test the target library, and GraphFuzz [19] is an API fuzzer that generates method sequences for library API functions and synthesizes harnesses for C/C++ programs with graph-based mutations.

The most related work would be AFGen [35], which introduces whole-function fuzzing to perform library fuzzing. While AFGen refine the fuzzing harness based on the constraints of the discovered crashes to reduce the false positive crashes. The commercial concolic testing engine CROWN 2.0 [26] automatically synthesizes fuzzing harness that assigns values to symbolic variables necessary for the target function execution from fuzzing inputs, targeting C programs.

All these works do not directly consider the validity of the function inputs, but rather focus on harness generation that generates valid function calls. The proposed approach is different from these works in that it focuses on the validity of the function inputs to filter out invalid inputs generated by the fuzzer. Several works mentioned that they can generate false positive crashes due to invalid API usage [19, 35], and the proposed function input validity estimation model approach can be used to filter out such invalid inputs.

Chapter 5. Conclusion

Regression testing can be done with function-level fuzzing, and patch-related function selection and identifying regression bugs are challenging problems. This study proposes to use system execution data, which is function coverage for patch-related function selection, and function input for identifying regression bugs. In patch-related function selection, the proposed regression relevance score shows higher acc@10 performance by 40%p compared to the state-of-the-art approach, demonstrating that considering both static call distance and dynamic function relevance is effective in selecting patch-related functions. In identifying regression bugs, the result shows that regression bugs can be found by analyzing only the top 20% of function-level crashes with high validity scores. These results highlight that system execution data effectively improves function-level regression fuzzing.

Future research will focus on improving function-level fuzzing to generate more diverse and valid function inputs. The default fuzzing algorithm saves the seed when exploring unseen branches, but it does not consider the validity of the inputs. If there are two seeds that have different validity but the same coverage, it may be better to keep the seed that has a higher validity score. Function-level fuzzing can be improved with a new fuzzing algorithm to find more valid function-level test cases.

Bibliography

[1] Onefuzz: A self-hosted fuzzing-as-a-service platform. https://github.com/microsoft/onefuzz accessed 20-December-2024.

[2] laf-intel, 2016. https://lafintel.wordpress.com, [Online].

[3] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. How long do vulnerabilities live in the code? a Large-Scale empirical measurement study on FOSS vulnerability lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 359–376, Boston, MA, August 2022. USENIX Association.

[4] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 975–985, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. Gpt-neox-20b: An open-source autoregressive language model, 2022. arXiv:2204.06745.

[6] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. Partition-based regression verification. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 302–311. IEEE Press, 2013.

[7] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 334–344, New York, NY, USA, 2013. Association for Computing Machinery.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Sadullah Canakci, Nikolay Matyunin, Kalman Graffi, Ajay Joshi, and Manuel Egele. Targetfuzz: Using darts to guide directed greybox fuzzers. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 561–573, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Oliver Chang, Jonathan Metzman, Max Moroz, Martin Barbella, and Abhishek Arya. Oss-fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz accessed 20-December-2024.

[11] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. arXiv:2107.03374.

[13] Stanley F Chen, Douglas Beeferman, and Roni Rosenfeld. Evaluation metrics for language models. 1998.

[14] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[15] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, page 197–211, New York, NY, USA, 1991. Association for Computing Machinery.

[16] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023. arXiv:2305.14314.

[17] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2440–2451, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[19] Harrison Green and Thanassis Avgerinos. Graphfuzz: library api fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1070–1081, New York, NY, USA, 2022. Association for Computing Machinery.

[20] Varun Gupta and Jitender Kumar Chhabra. Package coupling measurement in object-oriented software. *J. Comput. Sci. Technol.*, 24(2):273–283, March 2009.

[21] Gregory Hall, Wenyou Tao, and John Munson. Measurement and validation of module coupling attributes. *Software Quality Journal*, 13:281–296, 09 2005.

[22] Marc Heuse, Andrea Fiorald Heiko Eißfeld and, and Dominik Maier. Afl++, 2022. https://github.com/AFLplusplus/AFLplusplus, [Online; Downloaded Apr 24th, 2024].

[23] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[24] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287. USENIX Association, August 2020.

[25] Fred Jelinek, Robert L Mercer, Lalit R Bahl, and James K Baker. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America*, 62(S1):S63–S63, 1977.

[26] Hyunwoo Kim, Yunho Kim, and Moonzoo Kim. Improving applicability and usability of a concolic testing tool crown. *Journal of KIISE*, 45, 2018.

[27] Jeewoong Kim and Shin Hong. Poster: Bugoss: A regression bug benchmark for empirical study of regression fuzzing techniques. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 470–473, 2023.

[28] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. DAFL: Directed grey-box fuzzing guided by data dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4931–4948, Anaheim, CA, August 2023. USENIX Association.

[29] Yunho Kim, Yunja Choi, and Moonzoo Kim. Precise concolic unit testing of c programs using extended units and symbolic alarm filtering. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 315–326, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Yunho Kim, Shin Hong, and Moonzoo Kim. Target-driven compositional concolic testing with function summary refinement for effective bug detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 16–26, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, ICLR '17, 2017.

[32] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. Power: Program option-aware fuzzer for high bug detection ability. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 220–231, 2022.

[33] Gwangmu Lee, Duo Xu, Solmaz Salimi, Byoungyoung Lee, and Mathias Payer. Syzrisk: A change-pattern-based continuous kernel regression fuzzer. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, page 1480–1494, New York, NY, USA, 2024. Association for Computing Machinery.

[34] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *J. Syst. Softw.*, 23(2):111–122, November 1993.

[35] Yuwei Liu, Yanhao Wang, Xiangkun Jia, Zheng Zhang, and Purui Su. Afgen: Whole-function fuzzing for applications and libraries. In *2024 IEEE Symposium on Security and Privacy (SP)*, SP '24, pages 1901–1919, 2024.

[36] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, page 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[37] Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 235–245, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Paul Neculoiu, Maarten Versteegh, and Mihai Rotaru. Learning text similarity with siamese recurrent networks. In *Rep4NLP@ACL*, 2016.

[39] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*, 2023.

[40] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 815–816, New York, NY, USA, 2007. Association for Computing Machinery.

[41] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 504–515, New York, NY, USA, 2011. Association for Computing Machinery.

[42] Yu Rong, Yatao Bian, Tingyang Xu, Weiyang Xie, Ying Wei, Wenbing Huang, and Junzhou Huang. Self-supervised graph transformer on large-scale molecular data. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[43] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. arXiv:2308.12950.

[44] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, page 218–227, USA, 2008. IEEE Computer Society.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[46] Yuyang Wang, Jianren Wang, Zhonglin Cao, and Amir Barati Farimani. Molecular contrastive learning of representations via graph neural networks. *Nature Machine Intelligence*, 4(3):279–287, 2022.

[47] 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.

[48] Ji Yang, Xinyang Yi, Derek Zhiyuan Cheng, Lichan Hong, Yang Li, Simon Xiaoming Wang, Taibai Xu, and Ed H. Chi. Mixed negative sampling for learning two-tower neural networks in recommendations. In *Companion Proceedings of the Web Conference 2020*, WWW '20, page 441–447, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2169–2182, New York, NY, USA, 2021. Association for Computing Machinery.

Acknowledgments in Korean

가장 먼저, 연구실로 불러주시고 지도해주신 김문주 교수님께 감사드립니다. 기다려주시고 격려해주신 덕에 다사다난했던 석사과정을 잘 이겨낼 수 있었던 것 같습니다. 언제나 제가 잘 되기를 바라고, 조언을 아끼지 않았던 교수님의 따뜻한 지도가 있었기 때문에 지금까지 올 수 있었습니다. 교수님의 가르침을 잊지 않고, 앞으로도 꾸준히 성장하며 노력하겠습니다. 김윤호 교수님과 홍신 교수님께도 감사를 드립니다. 두 교수님들의 열정적인 모습이 제게 큰 귀감이 되었고, 지도를 받으면서 밤 늦게까지 같이 논문 내용을 고민하던 때가 오랫동안 기억에 남았습니다. 지금까지 부족하고 미성숙한 저를 이끌어주셔서 모두 감사합니다.

이아청 박사과정께도 감사를 드립니다. 제가 연구실 생활 하면서 불편하지 않도록 신경써주시고, 많은 일들을 맡아서 해주셨습니다. 혼자서도 많은 부담이 있었을 텐데, 도움을 요청하면 언제나 별다른 내색 없이 망설이지 않고 도와주셨습니다. Irfan Ariq, 강영빈, 양희찬 학생에게도 연구실에서 다방면으로 도움을 받았습니다. 기쁨과 슬픔을 나누면서 산책하거나 식사를 했던 때를 잊지 못할 것 같습니다.

김세민, 김형섭 학생에게도 감사하다는 말을 전하고 싶습니다. 대학원 기간 동안 같은 공간에서 생활하며 과정을 성공적으로 끝마칠 수 있도록 서로 아낌없이 격려해 주었습니다.

마지막으로, 오늘의 제가 있을 수 있도록 사랑으로 키워주신 아버지, 어머니, 할아버지, 할머니, 그리고 모든 가족들께 감사합니다. 가까이서 주기적으로 연락하며 도움을 언제든지 주셨던 부모님이 있었기에 여기까지 올 수 있었습니다. 제가 평생 보답할 수 있을 지 모를 정도로 가족들에게 과분한 사랑을 언제나 받아왔던 것 같습니다. 앞으로는 더 많은 것을 함께하며 은혜에 보답하겠습니다.

저의 이 작은 결실이 사랑하는 모든 분들께 조금이나마 보답이 되기를 바랍니다.

Curriculum Vitae

Name : Youngseok Choi

Date of Birth : July 27, 2000

E-mail : youngseok.choi00@gmail.com

Educations

2023. 2. – 2025. 2. M.S. Computer Science, KAIST

2019. 3. – 2023. 2. B.S. Computer Science and Mathematics - Double major, KAIST

2016. 3. – 2019. 2. Korea Science Academy of KAIST

Career

2024. 3. – 2024. 6. Teaching Assistant, CS458 Dynamic Analysis of Software Source Code

2023. 3. – 2023. 6. Teaching Assistant, CS458 Dynamic Analysis of Software Source Code

Publications

1. Ahcheong Lee, **Youngseok Choi**, Shin Hong, Yunho Kim, Kyutae Cho, and Moonzoo Kim, *ZigZag-Fuzz: Interleaved Fuzzing of Program Options and Files*, ACM Transactions on Software Engineering and Methodology, 2024.
2. **Youngseok Choi**, Ahcheong Lee, Hyoju Nam, Insub Lee, Namhoon Jung, Kyutae Cho, Moonzoo Kim, *Dynamic Unit State Data-driven False Alarm Filtering for Regression Unit Testing*, Journal of KIISE; Software and Applications, 36(11), 2024. (domestic journal)