Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master's Thesis

A Study on Support Tool of Solving Software Regression in JavaScript

JavaScript のソフトウェアリグレッションの解決に向けた支援ツールの研究

Yuefeng Hu
越峰 胡

Supervisor: Professor Shigeru Chiba

July 2024

Abstract

Software regression has been a persistent issue in software development. Although numerous techniques have been proposed to prevent regression from being introduced before release, few are available to address regression as it occurs post-release. Therefore, identifying the root cause of regression has always been a time-consuming and labor-intensive task. We aim to deliver automated solutions for solving regressions based on tracing. We present Bugfox, a trace-based analyzer that reports functions as the possible cause of regression in JavaScript. The idea is to generate runtime trace with instrumented programs, then extract the differences between clean and regression traces, and apply two heuristic strategies based on invocation order and frequency to identify the suspicious functions among differences. We evaluate our approach on 12 real-world regressions taken from the benchmark *BugsJS*. First strategy solves 6 regressions, and second strategy solves other 4 regressions, resulting in an overall accuracy of 83% on test cases. Notably, Bugfox solves each regression in under 1 minute with minimal memory overhead (<200 Megabytes). Our findings suggest Bugfox could help developers solve regression in real development.

概要

ソフトウェアの回帰は、ソフトウェア開発における持続的な問題となっています。リリース前に回帰が導入されるのを防ぐための技術が数多く提案されている一方で、リリース後に発生する回帰に対処するための手法は限られています。そのため、回帰の根本原因を特定することは、時間と労力を要する作業となっています。我々はトレースに基づく自動化ソリューションを提供することを目指しています。本論文では、回帰の可能性のある原因として関数を報告するトレースベースのアナライザーである Bugfox を提案します。このアイデアは、インストルメント化されたプログラムでランタイムトレースを生成し、クリーンなトレースと回帰トレースの違いを抽出し、呼び出し順序と頻度に基づく二つのヒューリスティック戦略を適用して、違いの中から疑わしい関数を特定するものです。我々のアプローチは、ベンチマーク BugsJS から採取した 12 の実際の回帰に対して評価されました。第一の戦略は 6 つの回帰を解決し、第二の戦略は他の 4 つの回帰を解決し、テストケースにおける全体的な精度は 83% となりました。特筆すべきは、Bugfox は各回帰を 1 分未満で解決し、メモリのオーバーヘッドは最小限（200 メガバイト未満）であることです。我々の研究結果は、Bugfox が実際の開発において開発者が回帰を解決するのに役立つ可能性があることを示唆しています。

Contents

Chapter 1

Introduction

Software regression, where previously functional software exhibits new faults after updates, remains a persistent challenge in software development. While many techniques focus on preventing regressions before release, fewer tools effectively address regressions post-release. Identifying the root cause of these regressions can be a tedious and time-consuming task for developers, often requiring extensive manual debugging. This issue is particularly pronounced in complex codebases with extensive commit histories and numerous interconnected components.

The ability to manage and rectify regressions efficiently is crucial for maintaining the quality and reliability of software systems. In today's fast-paced development environments, continuous integration and continuous deployment (CI/CD) practices are prevalent, and they emphasize the need for rapid detection and resolution of issues. However, traditional regression testing methods can fall short in these dynamic settings due to their reliance on manual processes and the inherent complexity of modern software applications. Therefore, there is a growing need for automated tools that can accurately and swiftly localize the causes of regressions.

In this thesis, we propose Bugfox, a trace-based analyzer designed to automate the localization of the root causes of software regressions in JavaScript. Our approach is grounded on the premise that discrepancies in execution traces between a properly functioning base program and a faulty, updated program can reveal the root cause of regressions. Based on this idea, we develop Bugfox, which traces both base and faulty programs by instrumenting their source code, extracts the differences of two traces, and reports suspicious functions among the differences based on two heuristic strategies. The first strategy focuses on the invocation relationships of function calls, aiming to identify the starting point of deviated behavior of the buggy program and return that one function call as the root cause of regression. The second strategy examines the frequency of function calls inside the differences of two traces, and returns top four functions with most occurrences as candidates. Bugfox collects those candidates and reports five candidates in total as possible root causes of regression.

We evaluate Bugfox on 12 real-world regressions from the BugsJS [1] benchmark, demonstrating its effectiveness and efficiency. The first strategy successfully resolves 6 regressions, and the second strategy resolves an additional 4 regressions, achieving an overall accuracy of 83% across test cases. Notably, Bugfox was able to solve each regression in under one minute with minimal memory overhead, making it a practical tool for real-world development environments.

The significance of Bugfox lies not only in its ability to accurately pinpoint the root causes of regressions but also in its practicality and ease of integration into existing development workflows. By automating the regression localization process, Bugfox reduces the time and effort required by developers, allowing them to focus on more critical tasks

and improving overall productivity. Additionally, the tool's minimal performance overhead ensures that it can be seamlessly integrated into CI/CD pipelines without adversely affecting build times or system performance.

In summary, the introduction of Bugfox offers a useful approach to automated regression testing. By utilizing execution traces and heuristic analysis, Bugfox helps address the challenge of identifying software regressions in JavaScript applications. The results from our evaluation show that Bugfox can assist developers in improving the process of regression testing. While Bugfox may not represent a major breakthrough, it validates the feasibility of automated tool that help solve regression in a more efficient and manageable way in real-world scenarios.

The structure of this thesis is as follows: In Chapter 2.1, we provide an overview of relevant background concepts, including different types of software testing, the nature of regressions, and various debugging techniques. Chapter 3 details the methodology used in the development and evaluation of Bugfox. This includes a description of the instrumentation process, the trace analysis algorithms, and the experimental setup. In Chapter 4, we present the experimental setup, results, and discussion. This chapter provides a detailed analysis of Bugfox's performance, including case studies and a discussion of its strengths and limitations. Finally, Chapter 5.1 concludes the thesis with a summary of findings and suggestions for future work.

Chapter 2

Localize the Cause of Regression

2.1 Background

Software regressions represent a significant challenge in software development, where a change or update inadvertently introduces new or previously resolved bugs. The detection and resolution of these regressions are critical for maintaining software quality and reliability. Initially, addressing regressions involved manual inspection and re-execution of test cases, a labor-intensive process that often led to delays and inefficiencies. Over time, the field of regression testing has evolved to incorporate automated testing tools designed to quickly execute extensive test suites and identify any new defects introduced by recent changes. This evolution has paralleled advancements in debugging techniques aimed at localizing the root cause of regressions. Tools and methodologies have been developed to enhance the precision of debugging processes, such as trace analysis and logging, which help developers pinpoint the exact source of defects. This progression highlights the ongoing effort to improve the efficiency and effectiveness of regression testing, reflecting the need for robust solutions in an increasingly complex software landscape.

To understand the context and significance of these advancements, it is essential to delve into the foundations of software testing, the nature of regressions, and the various debugging techniques employed in modern development. The following subsections will explore these areas in detail, beginning with an overview of software testing methodologies.

2.1.1 Software Testing

Software testing is a critical phase in the software development workflow that ensures the quality and functionality of the whole system. It involves the execution of software components to evaluate independent or integrated functionalities. The main goal of software testing is to find and fix defects, ensure that the software meets the expected requirements, and improve software quality. In this thesis, we talk about three common types of software testing: unit testing, integration testing, and system testing.

Unit Testing

Unit testing is a type of software testing where individual units or components of a software are tested in isolation from the rest of the application. Each unit, often a single function or method, is tested independently to verify that it behaves as expected. Unit tests are typically written by developers and are executed automatically, often as part of a continuous integration pipeline. They are essential for validating that the smallest parts of an application work correctly and for catching bugs early in the development process, and unit testing is one of the most commonly used testing methodology in real world. Unit testing frameworks, such as JUnit [2] for Java, NUnit [3] for .NET, and Mocha [4]

```
1 function sum(a, b) {
2   return a + b;
3 }
4 module.exports = sum;
```

Fig. 2.1: Function `sum`.

```
1 const sum = require('./sum');
2
3 describe("Test function sum", () => {
4   it('adds 1 + 2 should equal 3', () => {
5     assert.strictEqual(sum(1, 2), 3);
6   });
7 });
```

Fig. 2.2: Unit test code for function `sum`.

```
1 function addAndMultiply(a, b, c) {
2   return sum(a, b) * c;
3 }
4 module.exports = addAndMultiply;
```

Fig. 2.3: Function `addAndMultiply` using `sum`.

for JavaScript, provide tools and libraries to facilitate the creation and execution of unit tests.

Figure 2.1 and Figure 2.2 are examples of unit testing by using Mocha, a commonly used JavaScript testing framework. Figure 2.1 defines function `sum` which receives two values and return the sum of them. Figure 2.2 is the test code for function `sum`, line 5 verifies the the result of sum(1, 2) should be 3. Unit testing is a testing methodology which writes test code to verify the functionality of program components.

Integration Testing

Integration testing is a type of software testing where individual units or components are combined and tested as a group. The primary goal of integration testing is to identify issues that occur when different units interact with each other. Unlike unit tests, which test components in isolation, integration tests ensure that multiple components work together as intended. This type of testing can uncover issues related to interface mismatches, incorrect data sharing, and other integration problems that are not apparent in unit testing. Integration testing is often performed after unit testing and before system testing in the software development lifecycle.

Figure 2.3 and Figure 2.4 is a example of integration testing using Mocha. Figure 2.3 defines function `addAndMultiply` which relies on the previously defined `sum` function. Figure 2.4 is the test code for `addAndMultiply`, verifying that the combined functionality of summing and then multiplying works as expected. While this example demonstrates the basic concept of integration testing, real-world scenarios are often much more complex. In practice, integration testing might involve multiple interconnected components, databases, external APIs, and various configurations, all of which need to be validated to ensure the system operates correctly as a whole. The simplicity of this example serves to illustrate

```
const addAndMultiply = require('./addAndMultiply');
const sum = require('./sum');

describe("Test function addAndMultiply", () => {
  it('adds 1 + 2 and multiplies by 3 should equal 9', () => {
  assert.strictEqual(addAndMultiply(1, 2, 3), 9);
  });
});
```

Fig. 2.4: Integration test code for intermediate function `addAndMultiply`.

the fundamental approach, but actual applications would require more sophisticated test setups to handle the intricacies of real-world software systems.

System Testing

System testing is a type of software testing where a complete and integrated software system is tested as a whole. The objective of system testing is to validate the system's compliance with the specified requirements. This testing phase is crucial for identifying defects that may not be detected during unit or integration testing, as it tests the entire application in an environment that closely resembles production.

System tests evaluate the behavior of the entire system, including its interactions with external systems and components. This testing stage encompasses functional and non-functional testing types such as performance, usability, and security testing. System testing is usually performed by a dedicated testing team after integration testing and before acceptance testing.

2.1.2 Regression

Software regression refers to situations where previously functional software begins to exhibit new faults after updates or modifications. This challenge is prevalent in software development, especially as codebases grow and evolve. Regression testing is crucial for identifying these issues by ensuring that recent changes have not adversely affected existing functionality.

Regression Testing involves re-running previously executed tests to verify that new code changes have not introduced any unintended faults. Automated testing frameworks are commonly used to facilitate this process, enabling efficient and repeated execution of test cases. Causes of regression include:

1. Code Changes: Modifications to existing code may inadvertently affect other parts of the software.
2. Integration Issues: Merging new code with existing code can lead to conflicts and unintended side effects.
3. Dependency Updates: Changes to external libraries or frameworks might introduce compatibility issues.
4. Configuration Changes: Alterations in configuration or environment settings can impact software behavior.

Among those causes, one of the most common causes of regression bugs is the modification of existing code. When developers make changes to a codebase, whether to add new features, improve performance, or fix bugs, there is always a risk that these modifications can inadvertently affect other parts of the system. This unintentional impact can lead to

```
function formatDate(date) {
  // Original implementation formats date to 'MM/DD/YYYY'
  const month = String(date.getMonth() + 1).padStart(2, '0');
  const day = String(date.getDate()).padStart(2, '0');
  const year = date.getFullYear();
  return `${month}/${day}/${year}`;
}

function formatEvent(event) {
  return `Event: ${event.title} - Date: ${formatDate(new Date(
      event.date))}`;
}
```

Fig. 2.5: Original function `formateDate` and `formateEvent`.

```
describe('formatEvent', () => {
  it('should format an event with the date in MM/DD/YYYY', () =>
      {
    const today = new Date();
    const event = { title: 'Meeting', date: today.toISOString()
        };
    const todayFormatted = formatDate(today); // Assume
        formatDate returns MM/DD/YYYY by default

    const expected = `Event: Meeting - Date: ${todayFormatted}`;
    expect(formatEvent(event)).to.equal(expected);
  });
});
```

Fig. 2.6: Test code for function `formateEvent`.

the reappearance of previously resolved issues or introduce new problems in areas of the application that were not directly altered.

Figure 2.5 to Figure 2.7 are simple examples of regressions. Figure 2.5 is the code of function `formatDate` and `formatEvent`, where the former return formatted date and the latter return formatted report of event based on former function. Figure 2.6 is the test code of function `formatEvent`. These codes before modifications pass the test perfectly. After a certain period of time, the coder wants to exchange the order of month and day in date, so he changes the code of function `formatDate`, as shown in Figure 2.7. However, the coder forgets to update function `formatEvent` which based on `formatDate`. Now the test code in Figure 2.6 would fail, represents the unintentional destruction of existing functionality. Such software faults during the code modification is called a regression.

2.1.3 Debugging Techniques

Debugging is a crucial phase in the software development lifecycle, aimed at identifying, analyzing, and fixing bugs or defects in software. Several techniques are employed to facilitate this process, each offering distinct advantages depending on the nature of the problem. This section discusses some of the most common debugging techniques, including logging, using debuggers, profiling, and code reviews.

```
function formatDate(date, format = 'DD-MM-YYYY') {
  const day = String(date.getDate()).padStart(2, '0');
  const month = String(date.getMonth() + 1).padStart(2, '0');
  const year = date.getFullYear();
  return `${day}-${month}-${year}`;
}
```

Fig. 2.7: Updated function `formatDate` which introduces regression.

Logging

Logging is a fundamental debugging technique that involves recording information about the program's execution at various points during runtime. By inserting log statements in the code, developers can capture the state of the application, including variable values, execution flow, and function calls. This information is typically written to log files or system consoles, providing insights into the program's behavior and helping to trace the origin of bugs.

Logs can be categorized into different levels, such as DEBUG, INFO, WARN, ERROR, and FATAL, each representing the severity or type of information recorded. For instance, DEBUG logs provide detailed information useful for development and troubleshooting, while ERROR logs highlight critical issues that need immediate attention. Effective logging can greatly aid in diagnosing issues, particularly in complex systems where interactive debugging might be challenging.

Debugger

A debugger is a specialized tool that allows developers to pause the execution of a program, inspect its state, and step through the code line by line. This interactive approach provides a detailed view of the program's execution, including variable values, memory usage, and control flow. Modern integrated development environments (IDEs) often include built-in debuggers that offer features such as breakpoints, watchpoints, and stack traces.

Breakpoints enable developers to halt execution at specific lines of code, facilitating the examination of the program's state at critical points. Watchpoints monitor the value of variables or expressions, triggering a pause when changes occur. By stepping through the code, developers can observe how the program reaches its current state, making it easier to pinpoint the source of errors and understand the impact of code changes.

In case of debugging in JavaScript runtime environments, people always prefer the IDE graphical debuggers. Figure 2.8 shows the JavaScrit debugger of Visual Studio Code, which is a popular IDE [5]. The upper right part of the screen is the source code. A breakpoint is set as line 15 of this source code, and the executed program will be paused at this breakpoint. The bottom is the debug console where coder interacts with the program. The left side is several debug panels where coder can check the contents of variables, call stacks and breakpoints. These information could be used to find out the reason of unexpected behaviour through line-by-line execution and breakpoints.

In case of debugging website, built-in web broswer debuggers are oftenly used in real world. Figure 2.9 shows the built-in debugger of Chrome, which is a widely used web broswer around the world [6]. Similar to IDE debuggers, the screen of web broswer debuggers also consist of the panel of source code, variables, call stacks, breakpoints and graphical interaction interface.

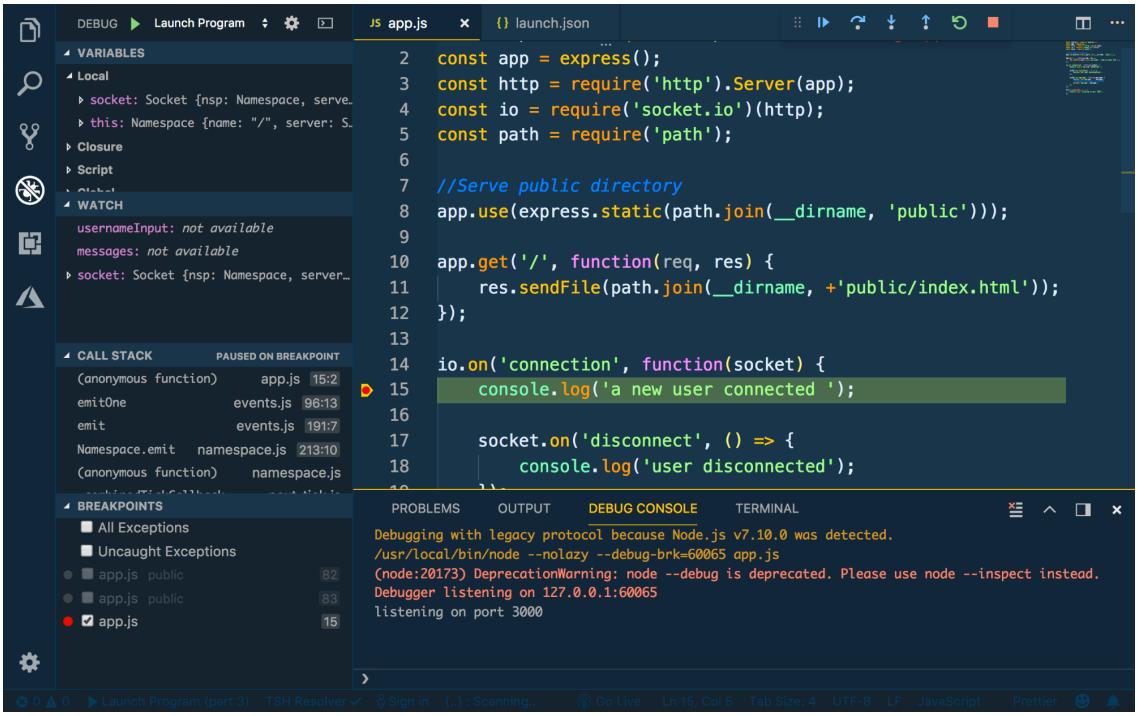

Fig. 2.8: Screenshot of JavaScritp debugger of Visual Studio Code.

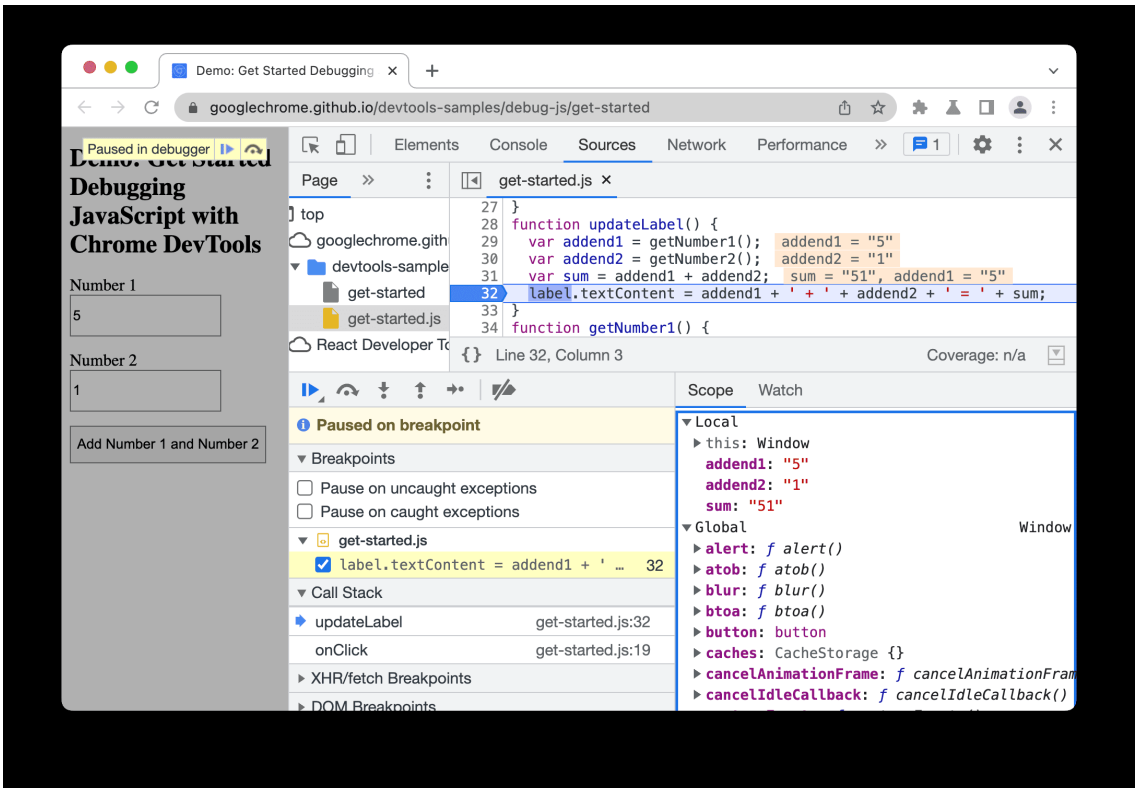

Fig. 2.9: Screenshot of built-in JavaScript debugger in Chrome.

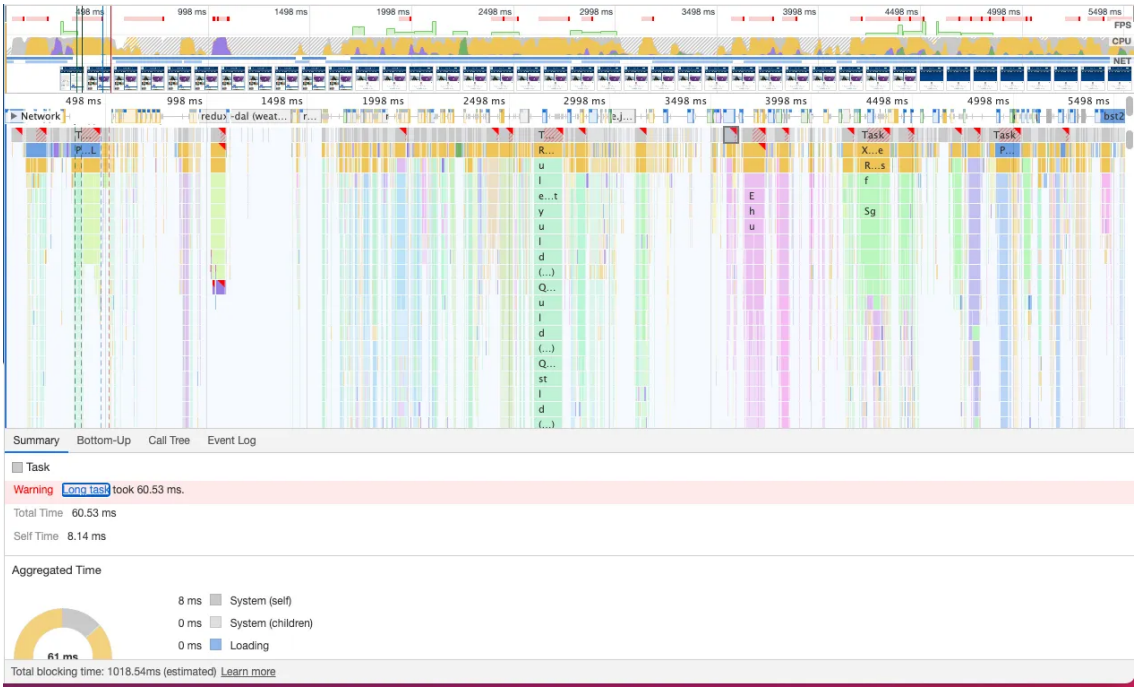

Fig. 2.10: Screenshot of Profiler in Chrome Devtools.

Profiling

Profiling involves analyzing the performance of a program to identify bottlenecks, resource consumption, and inefficient code paths. Profilers are tools that measure various aspects of program execution, such as CPU usage, memory allocation, and function call frequency. This information helps developers optimize performance and address issues related to resource management. Profilers can be classified into different types, including sampling profilers, which periodically check the program's state, and instrumented profilers, which add extra code to track execution details. By examining profiling data, developers can identify performance hotspots, evaluate the impact of code changes, and make informed decisions about optimizations.

Figure 2.10 is a screenshot of profiling engine in Chrome DevTools, a very popular profiler used in web development [7]. The center panel shows the performance of each function call, the x-axis represent the scale of execution time, each rectangle stands for a single function execution. The bottom of the screen includes the summary, call graph and logs of a program.

Code Review

Code review is a collaborative process in which developers examine each other's code to ensure quality, correctness, and adherence to coding standards. This technique involves reviewing code changes, providing feedback, and discussing potential improvements or issues. Code reviews can be conducted manually, through peer reviews, or using automated tools that analyze code for common issues.

The benefits of code reviews include catching bugs early in the development process, improving code readability and maintainability, and sharing knowledge among team members. By leveraging the collective expertise of the development team, code reviews help identify potential problems, enforce best practices, and enhance the overall quality of the software.

2.2 Related Work

The general methodology of existing regression debugging tools involves identifying, tracing, and analyzing behavioral changes between different software versions. These tools utilize dynamic analysis, change identification, machine learning, and historical data to pinpoint suspicious differences that might cause regression faults. However, the accuracy of these tools is often limited, and they typically do not provide direct contextual information, making it challenging for developers to quickly and effectively identify the root cause of regressions. Additionally, the automation level of these tools is constrained as they often require extra settings and configurations, which can be time-consuming and require specialized knowledge. These limitations underscore the need for more comprehensive, context-aware, and fully automated approaches to regression debugging. Next, we introduce related works that highlight the methods and limitations of existing tools, setting the stage for our approach.

Pastore et al. [8] introduce RADAR, a tool aimed at assisting developers in debugging regression issues in C/C++ software. RADAR combines change identification and dynamic analysis to detect and explain regression faults by highlighting suspicious behavioral differences between the base and upgraded software versions. The tool operates through three main steps:

1. Generation of Monitoring Scripts: RADAR generates scripts using the GDB debugger to monitor modified functions, their callers, and their callees, focusing on statements that remain unaltered in both versions for lightweight monitoring.
2. Data Collection: The tool collects behavioral data by executing test cases on both the base and upgraded versions separately, tracing variable values and sequences of executed statements, and distinguishing between passing and failing tests.
3. Analysis: RADAR generates models representing the legal behavior of the base version using Boolean expressions and Finite State Automata (FSA). It then compares these models with traces from the upgraded version to identify anomalies likely causing the regression.

Empirical results on both open-source and industrial software demonstrate RADAR's effectiveness, successfully identifying anomalies in 8 out of 10 regression faults. It provides valuable contextual information, aiding developers in quickly identifying and fixing regression faults. However, RADAR's applicability is limited by the constraints of low-level procedural languages, restricting universal tracing with complete data serialization. To explore universal tracing's feasibility for regressions, we develop a trace-based analyzer in JavaScript, capturing a comprehensive range of program information.

Maksimovic et al. [9] address a critical challenge in debugging large and complex digital systems by proposing an advanced framework for regression verification. Recognizing that debugging can constitute up to 70% of the design cycle and is often performed manually, the authors introduce a novel method that leverages machine learning and historical data to enhance the debugging process. Their framework focuses on automating the revision ranking process, which significantly improves the efficiency of identifying design changes that are likely to cause failures.

The proposed approach utilizes Support Vector Machine (SVM) classification to rank design revisions based on their likelihood of being the source of errors. Affinity Propagation (AP) clustering is then applied to group revisions with similar error profiles, further refining the ranking and prioritizing the most probable error sources. This method not only streamlines the debugging process by reducing the number of potential revisions

to examine but also improves the accuracy of failure diagnosis. Empirical results demonstrated the effectiveness of this framework, showing a 68% improvement in the accuracy of revision ranking compared to traditional brute force methods. Additionally, the runtime overhead of the proposed method is minimal, averaging just 4.631 seconds, making it a practical solution for real-world applications. This advancement in automated debugging addresses the inefficiencies of manual methods and offers a more systematic approach to handling large-scale digital designs.

By integrating machine learning with version control data, Maksimovic et al. provide a significant enhancement over existing debugging techniques, offering a more precise and efficient method for identifying and resolving errors in complex digital systems. This research motivates the construction of a regression bug dataset based on the version control system in this paper.

Unit testing frameworks have been implemented for various languages and platforms, including C [10], Java [2], JavaScript [11], Python [12], .NET [3, 13], Node.js [14, 4, 15] and React [16]. These frameworks are widely used in modern software development process, providing the possibilitiy to extend our approach to other programming languages.

Rosero [17] presents a survey, which reviews 15 years of software regression testing techniques from 2000 to 2015, covering 31 techniques from 25 papers. The paper classifies these techniques into minimization, selection, prioritization, and optimization categories. Key findings indicate that cost and fault detection efficiency are major considerations in technique evaluation. Most techniques were assessed in academic settings, with limited application in industrial contexts. The paper also highlights a trend towards incorporating regression testing in agile methodologies. The survey provides a comprehensive overview of the techniques' application domains, metrics used, strategies, and phases in the software development process. Our approach is inspired by its survey on regression test selection (RTS) techniques, where it describes the trend of testing techniques based on graphs, dependency relationships, historical data and heuristics.

2.3 Motivation with Examples

Regression has long been a persistent problem in software development process. Although detecting the regression is well studied in previous works [18, 17], localizing the cause of regression is still under exploration. In Figure 2.11, we show an example of a regression from a real-world project. The regression is from Hessian.js [19], which is the JavaScript implementation of Hessian binary web service protocol [20]. In Hessian.js, there are many utility functions implementing the encoding of raw data and decoding of binary data based on Hessian protocol. This example involves the release version 2.1.9 *(2b53d13)* (referred as the base version) and its subsequent commit *(a46fbc9)* (referred as the upgraded version). The green code with a minus `-` symbol is from the base version, and the red code with a plus `+` symbol is from the upgraded version.

The example implicitly introduces a regression in the upgraded version by incorrectly decoding the binary when raw data is `null` object. As shown in Figure 2.11, the condition of the `if` statement in base version is a regular expression that matches everything except strings with the word `this` followed by a dollar sign `$` and then one or more digits at the end of the string, such as `this$123` or `this$4567`. Therefore, the program would enter the `if` branch when `key` is a `null` object, and it would be encoded to an object with property `null` and value `'null'`, which is the correct implementation based on Hessian protocol when dealing with `null`. The upgraded version mistakenly adds a precondition to the `if` statement, leading to the program only entering the `if` branch when `key` is a string or number, and the `null` object would not be encoded in that scenario. To side with

the modification in Figure 2.11, the upgraded version also changes its corresponding unit testing function of decoding `null` object as illustrated in Figure 2.12, to pass all testing modules. Since the upgraded commit *(a46fbc9)* also involves new features and dozens of file changes, the developer merges this commit to the main tree without carefully reviewing this small change. This commit is blamed by other developers as a regression later and gets rolled back in a later fix *(37d19e7)*.

```
proto.readObject = function (withType) {
  ...
  key = this.read();
  ...
- if (!/^this\$\d+$/.test(key) {
+ t = typeof key;
+ if ((t === 'string' || t === 'number') && !/^this\$\d+$/.test(key) {
    result.$[key] = value;
  }
  ...
}
```

Fig. 2.11: Base (green) and upgraded (red) "buggy" version of function `proto.readObject` in decoder.js.

```
describe('map.test.js', function () {
  it('should decode successful when key is null', function () {
    var data = new Buffer([77, 116, 0, 0, 78, 83, 0, 4, 110, 117,
        108, 108, 122]);
    var rv = hessian.decode(data);
-   rv.should.eql(null:  'null');
+   rv.should.eql();
  });
}
```

Fig. 2.12: Base and upgraded "buggy" version of testing function related to foregoing `proto.readObject` function.

In most cases, addressing regressions still relies heavily on manual debugging by developers rather than relying on automated tools. This regression example may be perceived as straightforward to resolve, but imagine the regression is discovered after a long period of time along with numerous commits covering on the "regression" one, localizing the root cause of regression would become greatly difficult. In addition, in this example `proto.readObject` is just a middle-ware utility function used for implementing the decoding in hessian.js, while `hessian.decode` relates to hundreds of functions which all could possibly be regarded as the root cause of regression when developers inspect the limited output of failures reported by testing framework such as Mocha [4]. Given the complexity of pinpointing regression among extensive commits and intricate codebases, automated tools become indispensable and time-saving for efficiently resolving such tasks.

Among all testing techniques in modern test-driven development, unit testing as the most commonly used technique, still has limitations on addressing regressions and is not satisfying. Unit testing is also known as module testing, where individual units or components of a software application are tested in isolation to ensure they function correctly as

standalone units. However, with the tremendous expansion of modern software scale, it is considered burdensome and unpractical to design testing modules and test cases for every single unit and boundary condition; accordingly, testing modules always includes unit that directly or indirectly invokes other units, and regression often causes the infection of failures in this situation where one failed unit may affect the behavior of other units. As similar circumstance arises in integration testing, this chain reaction might result in vast failed test cases and bring greater difficulty to developers to locate the root cause of regression.

Chapter 3

Bugfox

In this thesis, we present Bugfox, which is a trace-based analyzer for localizing the cause of software regression in JavaScript. Our approach is based on the speculation that the root cause of regression could be detected inside the differences between execution traces of base program and buggy program, where the former represents the program that functions properly with a given unit test and the latter represents the updated program but fails on the same unit test. On the basis of this idea, we develop Bugfox, which traces both base and buggy programs by instrumenting their source code, extracts their differences, and reports suspicious functions among the differences based on two heuristic strategies. In addition, Git, as the state-of-the-art version control system, is responsible for maintaining and switching the version of source code in our approach. We assume that a buggy program is also committed, probably, to a debugging branch. So we below call base and buggy programs *base commit* and *buggy commit*, respectively.

Bugfox is designed to be a highly automated and user-friendly tool, all that is needed is a configuration file in *json* format, and developer does not have to write any extra code to make Bugfox running. Figure 3.2 shows an real example of that configuration file used in later experiment. Attribute `sourceFolder` means the relative path to home directory of source Git project, `generateFolder` means the relative path to home directory of generated folder that stores all outputs of Bugfox. Attribute `baseIgnoreFolder` and

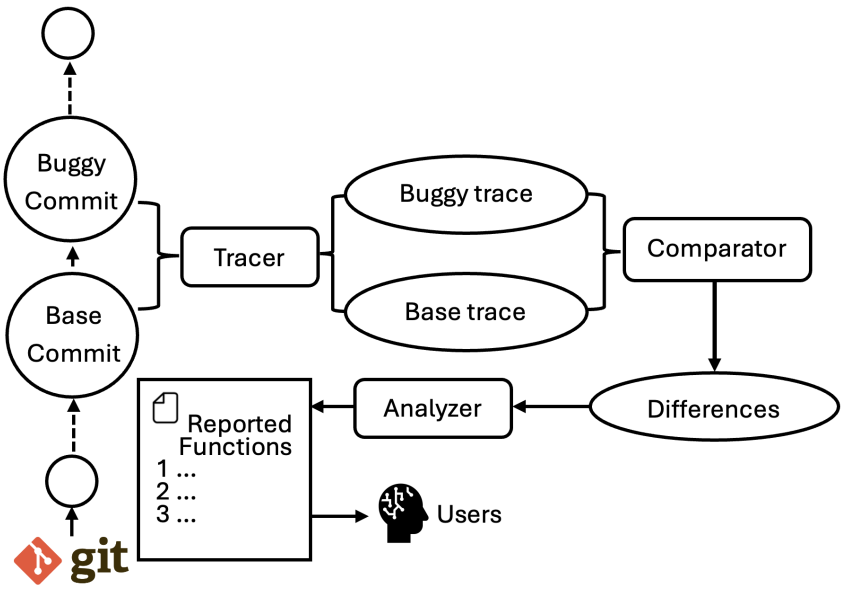

Fig. 3.1: Framework of Bugfox.

`newIgnoreFolder` stands for the folders or files being ignored in the code transformation in the project, often includes the path of external library. Attribute `baseCommitID` and `newCommitID` could be the SHA1 hash of one git commit, git tags, git branch or anything could be used by command `git switch -d $(argument)`, these attributes are used to indicate the version of base and buggy commit. Attribute `baseCommand` and `newCommand` are commands being executed for the test code that triggers the regression. After filling this configuration file, we only need one command to run Bugfox:

```
node Bugfox.js /path/to/config.json
```

Here the 'Bugfox.js' is the main program of Bugfox, and '/path/to/config.js' is the file path of our configuration file. After this simple command, Bugfox will automatically start tracing two versions of program during execution, comparing their traces, extracting the differences of two traces, analzying the suspicious functions inside the differences using two heuristic strategies and finally reporting the candidates of regression to users via command line. All results of experimental log and shell standard output will also be stored automatically into disk for later manual check.

```
{
  "sourceFolder": "BugJS/hessian",
  "generateFolder": "BugJS/hessian-result/Bug-2",
  "baseIgnoreFolder": [
    "node_modules"
  ],
  "newIgnoreFolder": [
    "node_modules"
  ],
  "baseCommitID": "Bug-2-base",
  "newCommitID": "Bug-2-new",
  "baseCommand": [
    "npm install",
    "NODE_OPTIONS=\"--max-old-space-size=64000\" ./node_modules/.
        bin/_mocha --timeout 99999 test/map.test.js"
  ],
  "newCommand": [
    "npm install",
    "NODE_OPTIONS=\"--max-old-space-size=64000\" ./node_modules/.
        bin/_mocha --timeout 99999 test/map.test.js"
  ]
}
```

Fig. 3.2: Example of that only needed configuration file *config.json* in Bugfox.

Compared to existing regression debugging tools, Bugfox offers significant improvements in both accuracy, contextual information and automation. While traditional tools often suffer from limited accuracy and lack direct context, Bugfox excels by providing detailed and direct contextual information about regression faults. This capability is achieved through comprehensive trace that allows for a clear understanding of how and where the program's behavior deviates between versions. Additionally, Bugfox's high level of automation minimizes the need for extensive configuration and manual setup. Unlike previous tools that require substantial extra settings or specialized knowledge, Bugfox operates with a single configuration file and a straightforward command, making it accessible and user-friendly. This ease of use, combined with its detailed contextual insights,

addresses many of the limitations of earlier methods, offering a more effective solution for identifying and resolving regression issues in software.

We give an overview of the workflow of Bugfox in Figure 3.1. The workflow could be divided into three independent process: tracing, comparing and analyzing. Section 3.1 to section 3.3 describes their specification.

3.1 Runtime tracing in JavaScript

As illustrated in Figure 3.1, the inputs of this workflow include two versions of source code, namely the base commit and the buggy commit. The base commit functions correctly, while the buggy commit introduces regressions in certain modules. Usually, the buggy commit resides chronologically after the base commit on the Git tree. Bugfox traces the execution of these programs and generates call graph [21] separately, namely a tree that represents the calling relationships between various functions within a program. Each node in the call graph represents a single function call, and the edges between nodes indicate the flow of invocation relationships between functions. An edge connects a node f to a node g when a function f is invoked, and a function g is invoked within the body of that function f.

A call-graph node has several attributes. One is an identifier to identify a called function. It is a concatenation of the path name of a source file, the names of functions surrounding the definition of the called function. For example, let function `onfinish` be defined in function `sendfile` in file `lib/response.js`. The identifier is this character string:

```
lib/response.js#Func@sendfile/Func@onfinish
```

Here, `#` and `/` are separators. `Func@` represents the kind of the function. BugFox supports seven kinds of functions: `Func` (normal function), `FuncVar` (function variable), `FuncExpr` (function expression), `method` (method in a class), `PropFunc` (property function), `ArrowMethod` (special arrow function as a method in a class), and `AnonFunc` (anonymous function). When a function does not have a name, the hash value of its body is used as its name. Figure 3.3 shows the example of each function type and how they will be named in our system.

Besides an identifier, a call-graph node has other attributes. It contains the hash value of the body of a called function, function arguments (the values of function parameters), the `this` object, and a return value. The value of `this` and the values of function parameters are obtained and recorded as node attributes twice, before and after the execution of the function body. There are all attributes inside one function node in the real implementation:

- `funcID`: The identifier of this function, includes their name with type and hash value of its function body.
- `index`: The index of this function node in the call graph.
- `caller`: The identifier of the caller of this function call.
- `beforeThis`: `this` arguments before its execution.
- `beforeArgs`: Function arguments before its execution.
- `afterThis`: `this` arguments after its execution.
- `afterArgs`: Function arguments after its execution.
- `returnVal`: Return value of this function execution.
- `callee`: The callees of this function call (functions invoked by it).

Those values about function specifications (`this`, arguments, return value) are repre-

```
1  // Func@func (normal function)
2  function func() {
3      console.log("This is a normal function.");
4  }
5
6  // FuncVar@func (function variable)
7  var func = function() {
8      console.log("This is a function variable.");
9  };
10
11 // FuncExpr@expr.func (function expression)
12 expr.func = function() {
13     console.log("This is a function expression.");
14 };
15
16 // method@func (method in a class)
17 class MyClass {
18     func() {
19         console.log("This is a method in a class.");
20     }
21 }
22
23 // PropFunc@func (property function)
24 const obj = {
25     func: function() {
26         console.log("This is a property function.");
27     }
28 };
29
30 // ArrowMethod@func (special arrow function as a method in a
       class)
31 class MyOtherClass {
32     func = () => {
33         console.log("This is an arrow function as a method in a
               class.");
34     };
35 }
36
37 // AnonFunc@af3y2c0 (anonymous function)
38 () => {
39     console.log("This is an anonymous function.");
40 };
```

Fig. 3.3: Code example with their name ID for each function type in JavaScript.

sented by character strings produced by the built-in library `JSON.stringify` in JavaScript with our extension. Although `JSON.stringify` can handle most objects correctly in JavaScript, there are still exceptions needed to be solved to achieve full coverage, includes function objects, circular objects and objects with accessors. Figure 3.4 shows the code example of these data types. The function objects will be ignored by `JSON.stringify`, circular objects are unable to be stringified, and objects with `get` accessors will cause unexpected extra function call and contaminate the call graph.

```
// Function objects
var func = function () { return 0; };
JSON.stringify(func); // return undefined

// Circular objects
const objA = {};
const objB = {};
objA.reference = objB;
objB.reference = objA;
JSON.stringify(objA); // runtime error, can't stringify circular
    object

// Objects with get accessor
const objWithAccessors = {
    _hiddenValue: 42,
    get hiddenValue() {
        return this._hiddenValue;
    }
};
JSON.stringify(objWithAccessors); // will call that hiddenValue
    function, causing unexpected extra behaviour
```

Fig. 3.4: Object types that could not be handled correctly by built-in `JSON.stringify`.

Figure 3.5 shows the code used to serialize every objects in JavaScript with our extension to built-in `JSON.stringify`. The function object will be wrapped into another normal object with certain attributes. The attributes which cause circulation in circular objects and `get` accessors will be pruning by using `WeakSet` to traverse all attributes and eliminate repetitive attributes which caused circulation and `get` accessors during serialization. The original object would not be modified during the serialization to guarantee the integrity of the program.

Bugfox generates call graphs for a base commit and a buggy commit taken from a git repository. For this aim, Bugfox conducts code transformation, which instruments the original source code in JavaScript to insert tracing statements that record runtime values as execution trace during program execution. To achieve that, we parse the source code into abstract syntax tree (AST) by using a JavaScript parse *acorn* [22], where abstract syntax tree is a data structure to represent the structure of a program or code snippet. Then we traverse the abstract syntax tree in depth first order using *estree walker* [23], a tool used for traversing an ESTree-compliant AST. During the traversal of AST, whenever we meet a node that represents a function declaration, we start to modify the subtree of this node and insert our tracing statements into their subtree.

However, before transforming these function declarations, we first regenerate a canonicalized form of function declaration using the unchanged function node with the help of a code generator *astring* in JavaScript, which is a tool to generate codes from abstract syntax tree. Then we calculate the hash value of the generated code as the hash value of a function declaration, which will be used to determine whether the code is different between two version. In this way, irrelevant code changes like linefeeds and spaces will be ignored, as long as the two codes share the same abstract syntax tree, they will be considered as identical function declaration. At this moment, the function identifier is also determined for each function declaration, in case it is an anonymous function, the hash value of its canonicalized function body will be used as the function identifier. After

```
const formatObj = (obj, seen = new WeakSet()) => {
  if (obj === undefined)
    return "-undefined-";
  else if (typeof obj !== 'object' && typeof obj !== 'function')
    return obj;
  else if (obj === null)
    return obj;

  if (seen.has(obj))
    return '[Circular]';
  seen.add(obj);

  const result = Array.isArray(obj) ? [] : {};
  for (const key of Object.getOwnPropertyNames(obj)) {
    const descriptor = Object.getOwnPropertyDescriptor(obj, key);
    if (descriptor && typeof descriptor.get === 'function')
      continue;

    const value = obj[key];
    result[key] = formatObj(value, seen);
  }
  seen.delete(obj);

  return result;
};
```

Fig. 3.5: Code used to serialize every object in JavaScript.

these process, we store the hash value and canonicalized function body separately into disk for later use.

Now we have to transform the source code and instrument our tracing statements into the AST. Figure 3.6 illustrates an example of code transformation. The process of transforming a JavaScript function involves several steps to enable detailed execution tracking. The following algorithm outlines these steps:

1. Wrap Original Function: Construct a new function with same name of origianl function, and move the original function inside the new function's body (becoming a internal function) and add a prefix to the original function name. This internal function retains the original functionality and will be used for actual execution while the outer function handles data serialization and tracing.
2. Insert Pre-execution Tracing Statement: Before invoking the original function, code is inserted to record the specification of the function call before its execution. The pseudocode `Record_before_exec(arg: [a, b], context: this)` demonstrates this step, where `arg` represents the serialized function arguments, and context captures the serialized `this` object. The data serialization of these values are automatically processed inside this pre-execution tracing statement.
3. Bind Context and Execute Function: The original function is executed within the correct context using `.bind(this)`. This ensures that the function behaves as intended in relation to original `this` context. The result of the function execution is captured in the variable `result`.
4. Insert Post-execution Tracing Statment: After the original function has executed, additional code records the specifications of this execution. The pseudocode

`Record_after_exec(arg: [a, b], context: this, ret: result)` illustrates this process. Here, `arg` and `context` are recorded again, and `result` represents the return value of the function. This post-execution data is serialized similarly to the pre-execution tracing statement.

5. Return Function Result: Return the result as it does not affect the behaviour of this function call.

With this proper wrapping of original function, the behaviour of this function execution does not change except tracing the information we need. The transformed function still maintain the original functionality. From other function's point of view, it still receive original arguments, have same function context, process the data as expected and return the original value. In this way, we gaurantee the code transformation does not influence the original functionality and cause unexpected behaviours. After code transformation, the instrumented program is run to record execution trace in memory. The recorded execution trace is finally written in a file before the program execution finishes.

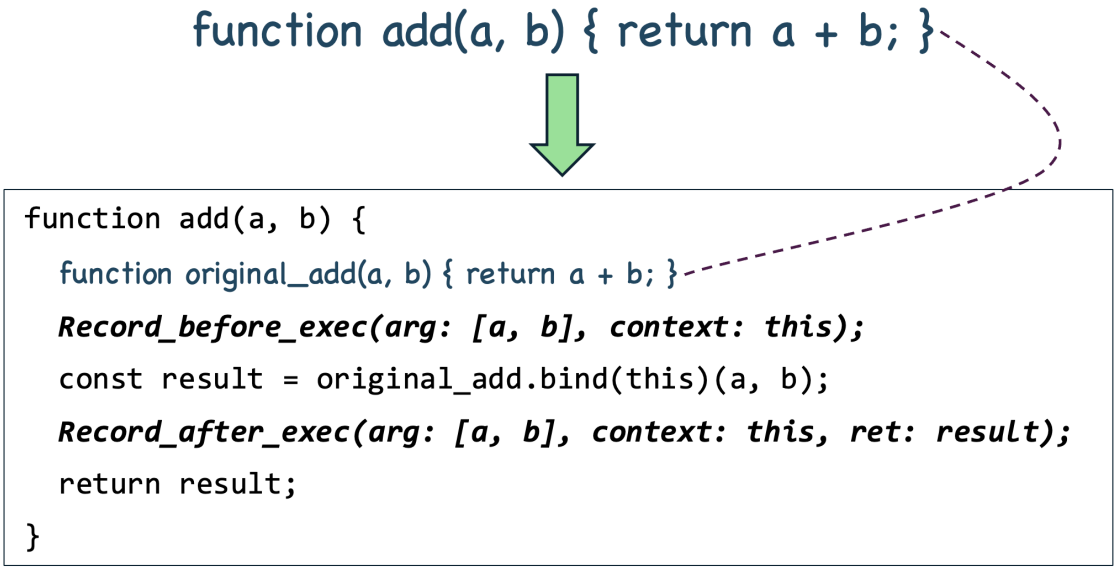

Fig. 3.6: Code transformation of a function definition (*italic* text represents pseudo-code for a tracing statement).

3.2 Differences extraction between call graphs

After obtaining the call graphs from tracing, we need to extract the ***differences*** between the base and the buggy traces. A difference is hereby defined as a pair of matching function node coming from two call graphs, but differing in the value of *arguments*, *this* object and *return value*. In order to accurately match up the homologous function node from two call graphs, we set up four prerequisites to guarantee two nodes represents same function call in their programs: 1) Same path (from root node) in the call graphs, 2) Both are n-th child node of same parent node, 3) Node with same function name, 4) All their previous sibling nodes are matched under same rules. Rule 1 and 2 guarantee two function calls have same invocation relationship in their program. Rule 3 ensures they stands for same function, so that we can analyze their differences legally. Last rule express that once two nodes are not matched, we stop the traversal of its remain sibling nodes since the unmatch indicates the invocation path inside its parent function execution has been

altered, therefore we can't guarantee the remaining sibiling nodes represents the same function call even if satisfy the first three prerequisites. Figure 3.7 shows an example of matching two call graphs. Function `zoo` is matched since they are both being invoked by `Func2` and they have same function name. Function `goo` and `foo` are not being matched since they have function name even they share the same index in the call graphs.

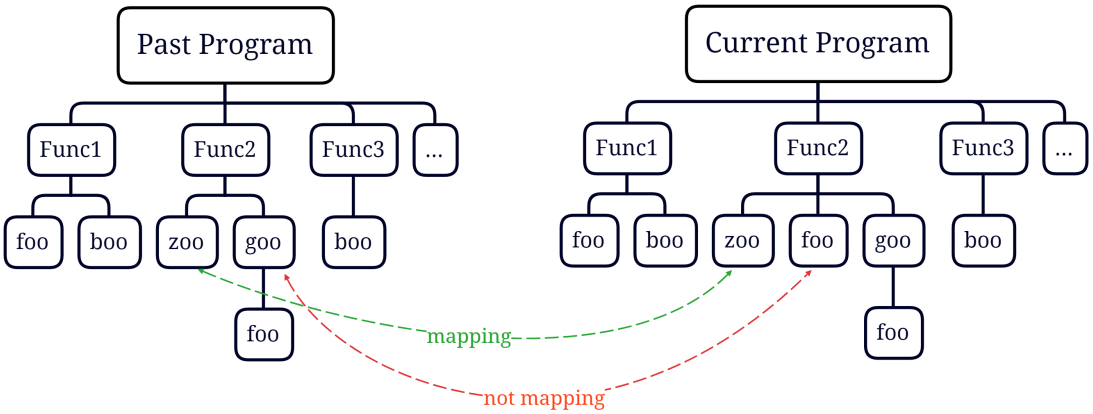

Fig. 3.7: Example of matching two call graphs.

After matching function nodes from two call graphs in depth first order (DFS), we compare their value of arguments, this object and return value before and after its execution. If any of them is different, we mark this pair of function node as a *difference* between two traces. In case these values are different before its execution, it indicates this deviation of its behaviour might be caused by its caller (passing different arguments or changing the context of one function call) instead of itself. Similarly, if these values are different only after its execution, it signifies the function itself is responsible for the different behaviour due to its code modification or inner functino calls. Furthermore, we compare the hash value of their function definitions to judge whether the function is being modified in the buggy commit. These results of comparison, including whether function specifications are being changed before its execution, and whether being changed after its execution, and whether function definition is modified in buggy commit, are collected for later analysis. With these information being collected, we can monitor the entrance and exit of each function call. As shown in Figure 3.8, three boolean values are created when comparing two matched function nodes:

- `isCodeChanged`: Whether the code of this function is being modified in current buggy program.
- `isBeforeChanged`: Whether those function specifications are different before its execution.
- `isAfterChanged`: Whether thoss function specifications and result are different after its execution.

3.3 Localization of the root causes

Finally, Bugfox localizes the root causes of regression by analyzing the differences extracted in the previous step. It adopts two heuristic strategies to prioritize the most likely candidates among the pairs of graph nodes marked as difference. One strategy focuses on the invocation order of function calls, and it chooses one candidate. The other focuses

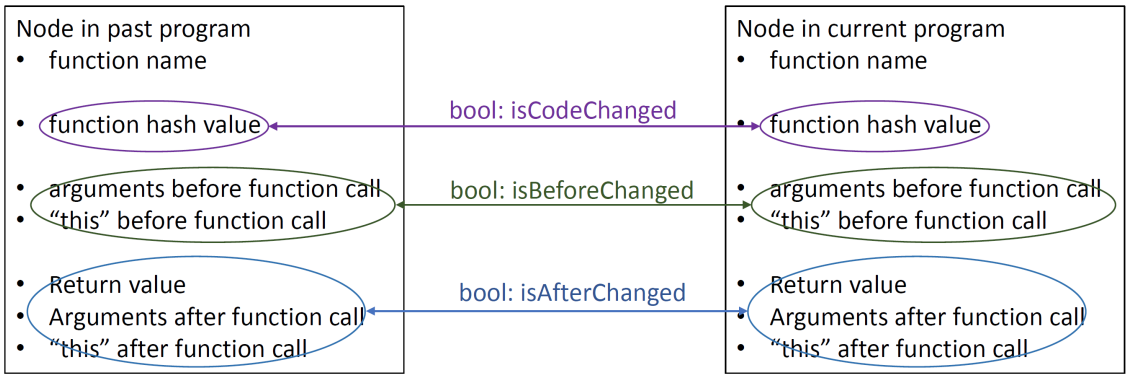

Fig. 3.8: Comparison of two function nodes.

on the frequency of function calls inside the differences, and it chooses four candidates. Bugfox collects those candidates and present them as a root cause of regression. Thus, Bugfox reports five candidates in total for the cause of regression.

3.3.1 First deepest function (FDF)

The first heuristic strategy identifies *the first deepest function* (FDF) among the differences, aiming to identify the starting point of deviated behavior of the buggy program. *Deepest functions* are the node pairs that are marked as difference but do not have descendants marked as difference in their subtrees. The function calls represented by those pairs show deviated behavior, but the nested calls to other functions from those calls do not show deviated behavior. We consider those *deepest function* calls as possible *root* causes of regression. Among all those deepest function calls, the chronologically first one could be regarded as the starting point of deviation of the buggy program, and thus it could be also considered as the beginning of the regression.

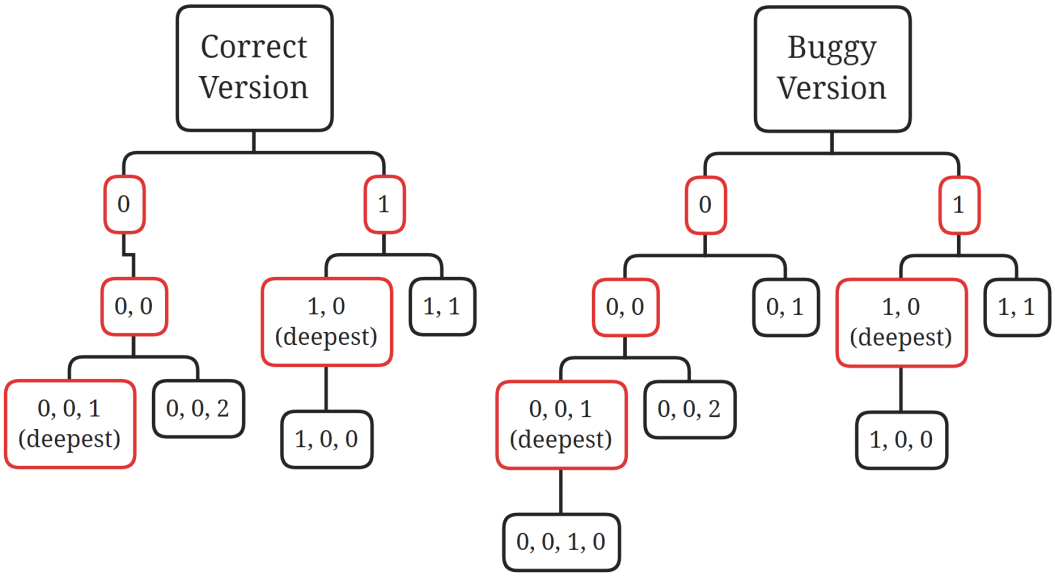

Fig. 3.9: Example of deepest functions in differences of two call graphs.

Figure 3.9 shows an example of deepest functions in differences of two call graphs. Each node represents one function execution, and the number inside of it stands for the index of this node in the call graph. After comparing, the different function executions will be marked, we label them with red color in this example. Among these differences, node `[0, 0, 1]` and `[1, 0]` are considered as deepest functions, and FDF strategy will return node `[0, 0, 1]` as the result since it is the chronologically first one in the call graph comparing to the latter. We think this function might be the beginning of the unexpected behaviour in the regression.

Bugfox selects the first deepest function (FDF) call as a candidate of the cause of regression. Before reporting it as a candidate, however, Bugfox checks how the selected pair of graph nodes is different from each other. As listed in Figure 3.1, Bugfox compares the attributes of the graph nodes of that pair. It compares the hash value of function bodies to check whether the source code changes between the base and buggy programs. It also compares the arguments (the values of the function parameters) and `this` object before and after the function execution. In Figure 3.1, the input states are those values before function execution while the output states are those values after function execution. If the results of the comparison match case 3, 4, or 5, then Bugfox reports that a candidate is the parent of the first deepest function call, which is the function that invokes the first deepest function call. For example, when a function f calls another function g and the call to g is the first deepest, the function f is a candidate of the cause of regression. If the results of the comparison match the other cases 1, 2, or 6, then Bugfox reports that a candidate is the function invoked by the first deepest function call. For the example above, the function g is a candidate.

Case	Is code changed	Is input state changed	Is output state changed	Speculation	Final reported function
1	no	no	yes	Deviated behaviour inside its execution	itself
2	yes	no	yes	Regard *code modification* as the cause of deviated behavior inside its execution	itself
3	no	yes	no	Rare condition where input state varies while output state remains the same	caller
4	yes	yes	no	Rare condition where input state varies while output state remains the same	caller
5	no	yes	yes	Possibly caused by the different arguments or `this` object before execution	caller
6	yes	yes	yes	Most complicated situation, need further inspection on the detail of difference	itself

Table 3.1: The decision table for the first deepest function strategy.

3.3.2 Top-n

The other heuristic strategy counts the occurrences of each function in the differences, then ranks them, and reports the top-n functions as candidates of the cause of regression. If multiple functions share the same counts, the function with the earliest invocation is being selected. Each pair of nodes marked as difference has an identifier as an attribute. This identifier represents the name of a called function. Bugfox checks this attribute to count the calls to each function. We assume that a function called in more node pairs marked as differences is the cause of regression with a greater likelihood.

Bugfox uses 4 as n for the Top-n strategy. It reports the top-4 functions as candidates of the cause of regression. We choose 4 to balance the accuracy and efficiency of this strategy. It is more likely that the selected candidates include the true cause of regression

when n is larger. However, the users of Bugfox must spend more time to investigate which candidate is a true cause.

The selection of the top-4 functions is based on simplifying the process of checking the results. Reporting too many candidates would require the user to invest a significant amount of time in investigating each one, which can be impractical and inefficient. On the other hand, reporting too few candidates increases the risk of missing the actual cause of regression. Thus, selecting n=4 provides a practical balance, ensuring a manageable number of candidates for the user to examine without overlooking potential causes.

Additionally, the Top-n strategy's robustness is enhanced by considering the context in which functions are invoked. Functions that are frequently called in different code paths or during various execution scenarios are given priority. This ensures that the strategy does not overly focus on a single execution path, which might be an anomaly, but rather provides a comprehensive view of potential regression causes.

By balancing between comprehensiveness and usability, the Top-n strategy aims to streamline the debugging process. It provides users with a manageable number of high-probability candidates to examine, thereby improving the efficiency of regression debugging in complex software systems. Future work may explore adaptive n-values based on the size and complexity of the codebase, further optimizing the strategy's effectiveness.

3.4 Limitation

Our methodology highly depends on the *differences* extracted from two call graphs. Predictably, whether two call graphs match well and whether valuable differences are collected directly affects the accuracy of our heuristic strategies. There are plenty situations might lead to the above dilemma, the most significant of which is *large scale refactoring*. Big scale refactoring always concerns the addition or deletion of functions and huge change of invocation order of function calls. In that case, one call graph might contain function calls that the other one does not have, and the matching of functions would be difficult due to the change of invocation order. Thereupon according to the rule of matching two call graphs, function calls which related to those refactorings would not be legally extracted into the differences. The tool can hardly solve the regression if those refactorings are responsible for the regression.

In addition, there are scenarios where our tool would fail even the desired *differences* are collected. Depending on the functionality of programs, there exists function calls related to *"mutable"* information such as port number, IP address, timestampe, or other randomly generated data. Differences extracted from such function calls might impede the analysis of our heuristic strategies in several ways. As for FDF strategy, once these differences appears chronologically earlier than the actual answer, FDF strategy might return these unrelated functions as the answer. In Top-n strategy, in case the frequency of such differences is higher than the actual answer, these unrelated functions will be ranked ahead the actual answer in n-candidates. Since these unrelated functions might result in false poisitive in our approach, we consider them as *noisy functions*.

Chapter 4

Experiments

In this thesis, we propose two research questions as follows.

RQ1. Can Bugfox be used to localize the root causes of regression?
RQ2. Can Bugfox meet the performance demands in real-world development?

In order to answer these research questions, we evaluate our system on 12 real-world regressions extracted from a JavaScript bug dataset *BugsJS* [1]. In this section, we will introduce our experiment settings, experiment results and the answers to the RQs. The experiments are conducted on a machine with an *11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz* CPU and 64GB RAM, and the operating system is Ubuntu 20.04.4 LTS.

4.1 Dataset and experiment settings

Currently, there is a lack of dataset especially focusing on regression bugs. In order for better evaluation, we screen and extract 12 regressions from a famous JavaScript bug dataset called *BugsJS* [1], which is a benchmark containing 453 real and manually-validated JavaScript bugs from 10 popular JavaScript server-side projects [24].

In order to extract these regressions, we take an automated approach to identify regression bugs systematically and efficiently, instead of manually inspecting each bug to determine if it is a regression. In *BugsJS*, each bug corresponds to a real Git commit and is labeled with a unique ID, its bug description, and its fix in real world that includes the update of the unit tests and source code. To note, the labeled Git commit may not include the cause of the bug, and it simply states the occurrence or detection of bug. To determine if a bug is a regression, we traverse the Git tree in reverse chronological order starting from the commit labeled as a bug, and execute the *updated* unit tests on every preceding commits. If we ever get a test failure in a commit followed by a test success in its parent commit, the bug could be considered as a regression since the same functionality has been correctly implemented before the code stops working. Meanwhile, we mark the commit that firstly introduces the bug as the *regression commit*.

We successfully recognize 12 bugs in *BugsJS* that could be considered as regressions. These bugs will be used to evaluate the accuracy and performance of Bugfox in this research. We collect their regression commits, and identify the functions being updated in a later fix as the root cause of the regressions. These functions are also considered as the ground truth to be compared with the results from Bugfox. In detail, 7 of these regression bugs come from Express [25], 3 of them come from ESLint [26], and 2 of them come from Hessian.js [19]. Express is a lightweight and flexible Node.js web application framework that simplifies building APIs and web servers with its robust features like routing, middleware support, and templating. ESLint is a JavaScript linting utility that helps developers find and fix problems in their code by enforcing coding standards, detecting errors, and

promoting consistent coding practices across projects. Hessian.js is a JavaScript library for encoding and decoding data using the Hessian binary protocol, enabling efficient and interoperable data transfer between JavaScript applications and systems that use the Hessian format. We give an overview of the collected bugs in Table 4.1. Note that the unit test of each bug case fails, and the generated log messages show only the failed assertion and the call stack of its test unit. Testing frameworks give no reasonable analysis or speculation for users to further understand and solve the regression. Finding the causes of regression is difficult without an extra tool like Bugfox.

Project	Bug-ID	Regression commit	Unit testing module
Express	1	f41d09a	test/app.options.js
Express	8	cf41a8f	test/app.use.js
Express	9	997a558	test/app.js
Express	13	31b2e2d	test/app.param.js
Express	16	c6e6203	test/res.redirect.js
Express	18	fb2d918	test/app.param.js
Express	27	7f04916	test/app.param.js
ESLint	10	a21dd32	tests/lib/config.js
ESLint	134	5266793	tests/lib/rules/no-useless-escape.js
ESLint	307	0f97279	tests/lib/rules/no-multi-spaces.js
hessian.js	2	a46fbc9	test/map.test.js
hessian.js	8	29f434e	test/v1.test.js

Table 4.1: List of 12 regressions extracted from *BugsJS*.

4.2 RQ1: Can Bugfox be used to localize the root causes of regression?

In each regression in Table 4.1, Bugfox will report 5 potential functions in total as the possible root causes of the regression, including one from FDF strategy and four from Top-n strategy. If the reported function aligns with the ground truth, we consider it as a success. Table 4.2 summarizes the results of applying Bugfox to localize the root cause of regression on the previously mentioned dataset. The 3rd column shows the fixed function for each regression in real world, considered as the ground truth in our experiment. The 4th column indicates whether the function reported by FDF strategy matches the ground truth. The 5th column shows whether the ground truth is included in the top-4 candidates reported by Top-n strategy with $n = 4$, and if included, which candidate matches the ground truth. Checkmark symbol ✓indicates the result of the strategy matches the expected result, while blank cell indicates the opposite.

As shown in Table 4.2, FDF strategy solves 6 regressions out of 12, and Top-n strategy solves 8 regressions out of 12. Among them, 4 regressions are shared by both strategies. In FDF strategy, all 7 cases from Express meet the case 5 in Table 3.1, which return the caller of first deepest function as the root cause. Hessian.js Bug-2 meet the case 1, and remaining 4 cases meet the case 2, these 5 cases return the first deepest function itself. By the integration of two strategies, Bugfox covers a total of 10 regressions, resulting in an overall accuracy of 83.3%.

We further verify the limitations of Bugfox in real scenario by investigating the failures of our experiments. It can be seen that the results of FDF strategy on Express project are not ideal. After inspecting the details of these cases, we find that Express, as a *de facto standard* server framework for Node.js, contains a massive amount of utility or middleware functions that include *unrelated* information such as port numbers, timestamps, IP addresses, or other randomly generated data. Such noise is liable to cause different behavior between two versions, leading to undesired differences of traces being collected and analyzed before the real suspicious difference. This situation happens in many regressions from Express project, resulting in the false reports of FDF strategy on Bug-9, 13, 16, 18, 27 from Express project. It also results in the failure of Top-n strategy on Bug-16 from Express project. However, Top-n strategy shows more tolerance since it provides more than one candidate.

Particularly, the complete failure of Bug-1 from Express project relates to the large-scale refactoring: the function that introduces the regression is added to the buggy commit for the first time. Hence, the base trace does not contain that function, resulting in the inability to extract meaningful differences and resolve the regression effectively.

Project	Bug-ID	Expected result	FDF strategy	Top-n strategy with n=4
Express	1	lib/router/index.js#FuncExpr@proto.handle		
	8	lib/application.js#FuncExpr@app.use	✓	2nd candidate
	9	lib/application.js#FuncExpr@app.use		3rd candidate
	13	lib/router/index.js#FuncExpr@proto.process_params		4th candidate
	16	lib/response.js#FuncExpr@res.redirect		
	18	lib/router/index.js#FuncExpr@proto.process_params		1st candidate
	27	lib/router/index.js#FuncExpr@proto.process_params		4th candidate
ESLint	10	lib/config.js#Class@Config/Method@constructor	✓	1st candidate
	134	lib/rules/no-useless-escape.js#PropFunc@create	✓	
	307	lib/rules/no-multi-spaces.js#PropFunc@create/PropFunc@Program	✓	
hessian.js	2	lib/v1/decoder.js#FuncExpr@proto.readObject	✓	4th candidate
	8	lib/utils.js#FuncExpr@exports.handleLong	✓	3rd candidate

Table 4.2: Experimental results of Bugfox on 12 regression test cases.

The following subsections detail the individual bug cases and how Bugfox was applied to each. These examples illustrate the specific contexts and output of each regression, providing a comprehensive understanding of Bugfox's performance and limitations.

4.2.1 Express Bug-1

```
var express = require('../')
  , request = require('supertest');

describe('OPTIONS', function(){
  it('should only include each method once', function(done){
    var app = express();

    app.del('/', function(){});
    app.get('/users', function(req, res){});
    app.put('/users', function(req, res){});
    app.get('/users', function(req, res){});

    request(app)
    .options('/users')
    .expect('GET,PUT')
    .expect('Allow', 'GET,PUT', done);
  })
})
```

Fig. 4.1: Test code to be executed in Express Bug-1.

Figure 4.1 is the test code being executed in Express Bug-1 case. It is designed to verify the behavior of the Express application when handling OPTIONS requests. The test sets up an Express app, defines various HTTP methods for specific routes, and then makes an OPTIONS request to the '/users' endpoint. It expects the 'Allow' header to list each HTTP method only once, specifically 'GET' and 'PUT'. This test is essential for ensuring that the application correctly handles HTTP method declarations without duplication.

```
OPTIONS
  1) should only include each method once

0 passing (31s)
1 failing

1) OPTIONS should only include each method once:
   Error: expected 'GET,PUT' response body, got 'GET,PUT,GET'
    at error (/home/icefox99/BugJS/express-result/Bug-1/project/
        express_run/node_modules/supertest/lib/test.js:235:13)
    at Test.assert (/home/icefox99/BugJS/express-result/Bug-1/
        project/express_run/node_modules/supertest/lib/test.js
        :180:21)
    at Server.assert (/home/icefox99/BugJS/express-result/Bug-1/
        project/express_run/node_modules/supertest/lib/test.js
        :132:12)
    at Object.onceWrapper (node:events:633:28)
    at Server.emit (node:events:519:28)
    at emitCloseNT (node:net:2279:8)
    at process.processTicksAndRejections (node:internal/process/
        task_queues:81:21)
```

Fig. 4.2: Output of regression bug in Express Bug-1.

Figure 4.2 is the shell output of test code above in Express Bug-1 regression commit. The test fails because the 'Allow' header incorrectly includes the 'GET' method twice instead of once, resulting in the response 'GET,PUT,GET' rather than the expected 'GET,PUT'. This failure indicates a regression in the handling of the OPTIONS method within the Express framework, where methods are not properly de-duplicated.

```
[TEST 69 & 134] test/app.options.js#AnonFunc@e7c2853/
    AnonFunc@d7809f6,d7809f6
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,c7befa3~~~~~~~~~~~~~~~~~~~~
[CODE_DIFF]
@@ -8,6 +8,10 @@
       debug('compile etag %s', val);
       this.set('etag fn', compileETag(val));
       break;
+    case 'query parser':
+      debug('compile query parser %s', val);
+      this.set('query parser fn', compileQueryParser(val));
+      break;
     case 'trust proxy':
       debug('compile trust proxy %s', val);
       this.set('trust proxy fn', compileTrust(val));


[DETAILS]
baseIndex: [69,0,0,0,0,0]
newIndex: [134,0,0,0,0,0]
caller: lib/application.js#FuncExpr@app.enable,a98a97d
isCodeChanged: true
isBeforeThisChanged: true
isBeforeArgsChanged: false
isAfterThisChanged: true
isAfterArgsChanged: false
isRetChanged: true

[ANALYSIS]
Most complicated situation, probably caused by the huge refactor
    of this function, please check all information includes its
    caller, callee and arguments.

...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,c7befa3~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.3: Report of First Deepest Function strategy in Express Bug-1.

Figure 4.3 presents the report generated by the First Deepest Function (FDF) strategy for identifying the root cause of the regression. It reports the 'app.set' function in 'lib/application.js' as the root cause of regression, which fails on this case while the right answer is 'proto.handle' function in 'lib/router/index.js'. After manually analyzing this regression case, we found that the function which causes the regression is not even included in the base commit, leading to nearly impossible to solve this case based on our methodology.

```
Function (Counts)
---------------------------------
lib/application.js#FuncExpr@app.set (4)
lib/application.js#AnonFunc@d7d4b9c/FuncExpr@app[method] (4)
lib/application.js#FuncExpr@app.enable (2)
lib/express.js#Func@createApplication (1)
```

Fig. 4.4: Report of Top-n strategy in Express Bug-1.

Figure 4.4 shows the output of Top-n strategy, which lists the functions most frequently

appearing in the execution trace differences between the base and buggy versions. Same as FDF strategy in this case, Top-n strategy also fails due to the big refactoring.

4.2.2 Express Bug-8

```
1  var after = require('after');
2  var express = require('..');
3  var request = require('supertest');
4
5  describe('app', function(){
6    it('should support empty string path', function (done) {
7      var app = express();
8
9      app.use('', function (req, res) {
10       res.send('saw ' + req.method + ' ' + req.url + ' through '
             + req.originalUrl);
11     });
12
13     request(app)
14     .get('/')
15     .expect(200, 'saw GET / through /', done);
16   })
17 })
```

Fig. 4.5: Test code to be executed in Express Bug-8.

Figure 4.5 is the test code being executed in the Express Bug-8 case. This test is designed to verify the behavior of the Express application when using an empty string path. It sets up an Express app, applies a middleware to an empty string path, and makes a GET request to the root endpoint. The test expects the response to confirm that the path was correctly handled as an empty string.

```
app
  1) should support empty string path

0 passing (714ms)
1 failing

1) app should support empty string path:
   TypeError: Router.use() requires callback function but got a [
       object String]
    at Bugfox_Original_94426d5 (/home/icefox99/BugJS/express-
        result/Bug-8/project/express_run/lib/router/index.js
        :389:17)
    at /home/icefox99/BugJS/express-result/Bug-8/project/
        express_run/lib/router/index.js:405:113
    at Array.forEach (<anonymous>)
  ....
    at /home/icefox99/BugJS/express-result/Bug-8/project/
        express_run/node_modules/mocha/lib/runner.js:308:7
    at next (/home/icefox99/BugJS/express-result/Bug-8/project/
        express_run/node_modules/mocha/lib/runner.js:246:23)
    at Immediate.<anonymous> (/home/icefox99/BugJS/express-result
        /Bug-8/project/express_run/node_modules/mocha/lib/runner.
        js:275:5)
    at process.processImmediate (node:internal/timers:478:21)
```

Fig. 4.6: Output of regression bug in Express Bug-8.

Figure 4.6 shows the shell output of the test code above in the Express Bug-8 regression commit. The test fails because the Router.use() method incorrectly treats the empty string path as invalid, resulting in a TypeError. This indicates a regression in the middleware path handling within the Express framework.

```
[TEST 137 & 137] test/app.use.js#AnonFunc@52ae34c/
    AnonFunc@1bfdd54,1bfdd54
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/utils.js#FuncExpr@exports.
    flatten,f7a6daa~~~~~~~~~~~~~~~~~~~~
[CODE]
function (arr, ret) {
  ret = ret || [];
  var len = arr.length;
  for (var i = 0; i < len; ++i) {
    if (Array.isArray(arr[i])) {
      exports.flatten(arr[i], ret);
    } else {
      ret.push(arr[i]);
    }
  }
  return ret;
}

[DETAILS]
index: [137,1,0]
caller: lib/application.js#FuncExpr@app.use,c2a6b41
isCodeChanged: false
isBeforeThisChanged: false
isBeforeArgsChanged: true
isAfterThisChanged: false
isAfterArgsChanged: true
isRetChanged: true

[ANALYSIS]
Probably caused by the different arguments, please check its
    CALLER "lib/application.js#FuncExpr@app.use,c2a6b41" and the
    different arguments that passed to this function.

...

[COMPARISON]
BASE->beforeThis: null
NEW-->beforeThis: null
---------------------------------
BASE->beforeArgs: [[{"length":2,"name":"","arguments":null,"
    caller":null,"prototype":{"constructor":"[Circular]"}}]]
NEW-->beforeArgs: [["",{"length":2,"name":"","arguments":null,"
    caller":null,"prototype":{"constructor":"[Circular]"}}]]
DIFF:
@@ -1,5 +1,6 @@
 [
   [
+    "",
     {
       "length": 2,
       "name": "",

...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/utils.js#FuncExpr@exports.
    flatten,f7a6daa~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.7: Report of First Deepest Function strategy in Express Bug-8.

Figure 4.7 presents the report generated by the First Deepest Function (FDF) strategy for identifying the root cause of the regression. It reports the 'app.use' function in 'lib/ap-

plication.js' as the root cause of the regression. The analysis indicates that the regression is probably caused by different arguments passed to from 'app.use' to anonymous function '1bfdd54'. FDF strategy successfully identifies the root cause of regression in this case.

```
Function (Counts)
-----------------------------------
lib/utils.js#FuncExpr@exports.flatten (2)
lib/application.js#FuncExpr@app.use (1)
lib/application.js#FuncExpr@app.use/AnonFunc@497a64c (1)
lib/router/index.js#FuncExpr@proto.use (1)
```

Fig. 4.8: Report of Top-n strategy in Express Bug-8.

Figure 4.8 shows the output of the Top-n strategy in Express Bug-8 case, listing the functions most frequently appearing in the execution trace differences between the base and buggy versions. Similar to the FDF strategy, the root cause of regression 'app.use' is also included in the four candidates of Top-n strategy.

4.2.3 Express Bug-9

```
1  var assert = require('assert')
2  var express = require('..')
3  var request = require('supertest')
4
5  describe('app.mountpath', function(){
6    it('should return the mounted path', function(){
7      var admin = express();
8      var app = express();
9      var blog = express();
10     var fallback = express();
11
12     app.use('/blog', blog);
13     app.use(fallback);
14     blog.use('/admin', admin);
15     fallback.use('/admin', admin);
16     blog.use(fallback);
17
18     admin.mountpath.should.equal('/admin');
19     app.mountpath.should.equal('/');
20     blog.mountpath.should.equal('/blog');
21     fallback.mountpath.should.equal('/');
22   })
23 })
```

Fig. 4.9: Test code to be executed in Express Bug-9.

Figure 4.9 is the test code being executed in the Express Bug-9 case. This test is designed to verify the behavior of the Express application when using the mountpath property to retrieve the mounted path of different Express instances. The test sets up multiple Express apps and mounts them in various configurations. It then checks if the mountpath property correctly reflects the paths at which the apps are mounted.

```
0 passing (4s)
1 failing

1) app.mountpath should return the mounted path:
   AssertionError [ERR_ASSERTION]: expected { [Function: app]
   init: [Function],
   defaultConfiguration: [Function],
   lazyrouter: [Function],
   handle: [Function],
   ...
    at prop.<computed> (/home/icefox99/BugJS/express-result/Bug
        -9/project/express_run/node_modules/should/lib/should.js
        :61:14)
    at context.Bugfox_Original_f859fc4 (/home/icefox99/BugJS/
        express-result/Bug-9/project/express_run/test/app.js
        :21:35)
    at context.<anonymous> (/home/icefox99/BugJS/express-result/
        Bug-9/project/express_run/test/app.js:28:113)
    at callFn (/home/icefox99/BugJS/express-result/Bug-9/project/
        express_run/node_modules/mocha/lib/runnable.js:223:21)
    at Runnable.run (/home/icefox99/BugJS/express-result/Bug-9/
        project/express_run/node_modules/mocha/lib/runnable.js
        :216:7)
    at Runner.runTest (/home/icefox99/BugJS/express-result/Bug-9/
        project/express_run/node_modules/mocha/lib/runner.js
        :373:10)
    at /home/icefox99/BugJS/express-result/Bug-9/project/
        express_run/node_modules/mocha/lib/runner.js:451:12
    at next (/home/icefox99/BugJS/express-result/Bug-9/project/
        express_run/node_modules/mocha/lib/runner.js:298:14)
    at /home/icefox99/BugJS/express-result/Bug-9/project/
        express_run/node_modules/mocha/lib/runner.js:308:7
    at next (/home/icefox99/BugJS/express-result/Bug-9/project/
        express_run/node_modules/mocha/lib/runner.js:246:23)
    at Immediate._onImmediate (/home/icefox99/BugJS/express-
        result/Bug-9/project/express_run/node_modules/mocha/lib/
        runner.js:275:5)
    at process.processImmediate (node:internal/timers:478:21)
```

Fig. 4.10: Output of regression bug in Express Bug-9.

Figure 4.10 shows the shell output of the test code above in the Express Bug-9 regression commit. The test fails because the mountpath property does not correctly return the expected mounted paths. This failure indicates a regression in how the Express framework handles the mountpath property for mounted applications.

```
[TEST 137 & 137] test/app.js#AnonFunc@6a055ca/AnonFunc@f859fc4,
    f859fc4
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,c7befa3~~~~~~~~~~~~~~~~~~~~
[CODE]
function (setting, val) {
  if (arguments.length === 1) {
    return this.settings[setting];
  }
  this.settings[setting] = val;
  switch (setting) {
    case 'etag':
      debug('compile etag %s', val);
      this.set('etag fn', compileETag(val));
      break;
    case 'trust proxy':
      debug('compile trust proxy %s', val);
      this.set('trust proxy fn', compileTrust(val));
      break;
  }
  return this;
}

[DETAILS]
index: [137,0,0,0,0,0]
caller: lib/application.js#FuncExpr@app.enable,a98a97d
isCodeChanged: false
isBeforeThisChanged: true
isBeforeArgsChanged: false
isAfterThisChanged: true
isAfterArgsChanged: false
isRetChanged: true

[ANALYSIS]
Probably caused by the different arguments, please check its
    CALLER "lib/application.js#FuncExpr@app.enable,a98a97d" and
    the different arguments that passed to this function.

...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,c7befa3~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.11: Report of First Deepest Function strategy in Express Bug-9.

Figure 4.11 presents the report generated by the First Deepest Function (FDF) strategy for identifying the root cause of the regression. It reports the 'app.enable' function in 'lib/application.js' as the root cause of the regression. FDF strategy fails on Express Bug-9 case. After inspecting the call graphs and extracted differences, we find that there are many noisy functions affected the matching of two call graphs, which lead to the unexpected behaviour not being matched up between two version and extracted into the differences.

```
Function (Counts)
---------------------------------
lib/application.js#FuncExpr@app.set (46)
lib/application.js#FuncExpr@app.enabled (6)
lib/application.js#FuncExpr@app.use (5)
lib/application.js#FuncExpr@app.lazyrouter (5)
```

Fig. 4.12: Report of Top-n strategy in Express Bug-9.

Figure 4.12 shows the output of the Top-n strategy in Express Bug-9 case, listing the functions most frequently appearing in the execution trace differences between the base and buggy versions. Unlike FDF strategy, the root cause of regression 'app.use' is included in the four candidates of Top-n strategy. This frequency-based heuristic strategy has the chance to overcome the influence of noisy functions and bad matching, which is the disadvantage of FDF strategy.

4.2.4 Express Bug-13

```
1  var express = require('../')
2    , request = require('supertest');
3
4  describe('app', function(){
5    describe('.param(name, fn)', function(){
6      it('should support altering req.params across routes',
           function(done) {
7        var app = express();
8
9        app.param('user', function(req, res, next, user) {
10         req.params.user = 'loki';
11         next();
12       });
13
14       app.get('/:user', function(req, res, next) {
15         next('route');
16       });
17       app.get('/:user', function(req, res, next) {
18         res.send(req.params.user);
19       });
20
21       request(app)
22       .get('/bob')
23       .expect('loki', done);
24     })
25   })
26 })
```

Fig. 4.13: Test code to be executed in Express Bug-13.

Figure 4.13 is the test code being executed in the Express Bug-13 case. This test is designed to verify the behavior of the Express application when using the param method to alter request parameters across different routes. The test sets up an Express app and

defines a parameter middleware that changes the value of the user parameter. It then defines two routes that both use the altered parameter.

```
0 passing (11s)
1 failing

1) app .param(name, fn) should support altering req.params across
     routes:
   Error: expected 'loki' response body, got 'Cannot GET /bob\n'
    at error (/home/icefox99/BugJS/express-result/Bug-13/project/
        express_run/node_modules/supertest/lib/test.js:227:13)
    at Test.assert (/home/icefox99/BugJS/express-result/Bug-13/
        project/express_run/node_modules/supertest/lib/test.js
        :172:21)
    at /home/icefox99/BugJS/express-result/Bug-13/project/
        express_run/node_modules/supertest/lib/test.js:125:10
    at Request.callback (/home/icefox99/BugJS/express-result/Bug
        -13/project/express_run/node_modules/superagent/lib/node/
        index.js:728:30)
    at Test.<anonymous> (/home/icefox99/BugJS/express-result/Bug
        -13/project/express_run/node_modules/superagent/lib/node/
        index.js:135:10)
    at Test.emit (node:events:520:28)
    at IncomingMessage.<anonymous> (/home/icefox99/BugJS/express-
        result/Bug-13/project/express_run/node_modules/superagent
        /lib/node/index.js:886:12)
    at IncomingMessage.emit (node:events:532:35)
    at endReadableNT (node:internal/streams/readable:1696:12)
    at process.processTicksAndRejections (node:internal/process/
        task_queues:82:21)
```

Fig. 4.14: Output of regression bug in Express Bug-13.

Figure 4.14 shows the shell output of the test code above in the Express Bug-13 regression commit. The test fails because the parameter alteration middleware does not correctly modify the user parameter as expected. Instead of returning the expected 'loki', it results in a 'Cannot GET /bob' error, indicating a regression in how the Express framework handles parameter alteration.

```
[TEST 131 & 131] test/app.param.js#AnonFunc@a2a917c/
    AnonFunc@2a6bfeb/AnonFunc@434b76a,434b76a
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,a1fe05f~~~~~~~~~~~~~~~~~~~~
[CODE]
function (setting, val) {
  if (1 == arguments.length) {
    return this.settings[setting];
  } else {
    this.settings[setting] = val;
    return this;
  }
}

[DETAILS]
index: [131,0,0,0,0,0]
caller: lib/application.js#FuncExpr@app.enable,a98a97d
isCodeChanged: false
isBeforeThisChanged: true
isBeforeArgsChanged: false
isAfterThisChanged: true
isAfterArgsChanged: false
isRetChanged: true

[ANALYSIS]
Probably caused by the different arguments, please check its
    CALLER "lib/application.js#FuncExpr@app.enable,a98a97d" and
    the different arguments that passed to this function.

...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,a1fe05f~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.15: Report of First Deepest Function strategy in Express Bug-13.

Figure 4.15 presents the report generated by the First Deepest Function (FDF) strategy for identifying the root cause of the regression. It reports the app.enable function in lib/application.js as the root cause of the regression. Similar to Express Bug-9, the FDF strategy fails in the Express Bug-13 case due to the influence of noisy functions and bad matching. The differences in the call graphs between the base and buggy versions lead to an incorrect identification of the root cause.

```
Function (Counts)
-----------------------------------
lib/application.js#FuncExpr@app.set (11)
lib/application.js#FuncExpr@app.enabled (3)
lib/application.js#AnonFunc@4c5011c/FuncExpr@app[method] (3)
lib/router/index.js#FuncExpr@proto.process_params (3)
```

Fig. 4.16: Report of Top-n strategy in Express Bug-13.

Figure 4.16 shows the output of the Top-n strategy in the Express Bug-13 case, listing the functions most frequently appearing in the execution trace differences between the base and buggy versions. Unlike the FDF strategy, the Top-n strategy successfully includes the root cause of the regression, 'proto.process_params', in its list of candidates.

4.2.5 Express Bug-16

```
var express = require('../')
  , request = require('supertest');

describe('res', function(){
  describe('when accepting text', function(){
    it('should include the redirect type', function(done){
      var app = express();

      app.use(function(req, res){
        res.redirect(301, 'http://google.com');
      });

      request(app)
      .get('/')
      .set('Accept', 'text/plain, */*')
      .expect('Content-Type', /plain/)
      .expect('Location', 'http://google.com')
      .expect(301, 'Moved Permanently. Redirecting to http://
          google.com', done);
    })
  })
})
```

Fig. 4.17: Test code to be executed in Express Bug-16.

Figure 4.17 is the test code being executed in the Express Bug-16 case. This test is designed to verify the behavior of the Express application when handling redirects while accepting text. The test sets up an Express app that issues a 301 redirect to 'http://google.com'. It then checks if the response includes the correct redirect type in the response body.

```
0 passing (26s)
1 failing

1) res when accepting text should include the redirect type:
   Error: expected 'Moved Permanently. Redirecting to http://
      google.com' response body, got 'Moved Permanently.
      Redirecting to 301'
    at error (/home/icefox99/BugJS/express-result/Bug-16/project/
       express_run/node_modules/supertest/lib/test.js:235:13)
    at Test.assert (/home/icefox99/BugJS/express-result/Bug-16/
       project/express_run/node_modules/supertest/lib/test.js
       :180:21)
    at Server.assert (/home/icefox99/BugJS/express-result/Bug-16/
       project/express_run/node_modules/supertest/lib/test.js
       :132:12)
    at Object.onceWrapper (node:events:634:28)
    at Server.emit (node:events:520:28)
    at emitCloseNT (node:net:2321:8)
    at process.processTicksAndRejections (node:internal/process/
       task_queues:81:21)
```

Fig. 4.18: Output of regression bug in Express Bug-16.

Figure 4.18 shows the shell output of the test code above in the Express Bug-16 regression commit. The test fails because the response body does not include the expected redirect type. Instead of returning 'Moved Permanently. Redirecting to http://google.com', it returns 'Moved Permanently. Redirecting to 301', indicating a regression in how the Express framework handles redirects when the client accepts text. It could be seen that the status code is wrongly passed to result instead of expected url.

```
[TEST 137 & 137] test/res.redirect.js#AnonFunc@4a6a13e/
    AnonFunc@44dad68/AnonFunc@5bb7f9a,5bb7f9a
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,c7befa3~~~~~~~~~~~~~~~~~~~~
[CODE]
function (setting, val) {
  if (arguments.length === 1) {
    return this.settings[setting];
  }
  this.settings[setting] = val;
  switch (setting) {
    case 'etag':
      debug('compile etag %s', val);
      this.set('etag fn', compileETag(val));
      break;
    case 'trust proxy':
      debug('compile trust proxy %s', val);
      this.set('trust proxy fn', compileTrust(val));
      break;
  }
  return this;
}

[DETAILS]
index: [137,0,0,0,0,0]
caller: lib/application.js#FuncExpr@app.enable,a98a97d
isCodeChanged: false
isBeforeThisChanged: true
isBeforeArgsChanged: false
isAfterThisChanged: true
isAfterArgsChanged: false
isRetChanged: true

[ANALYSIS]
Probably caused by the different arguments, please check its
    CALLER "lib/application.js#FuncExpr@app.enable,a98a97d" and
    the different arguments that passed to this function.

...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,c7befa3~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.19: Report of First Deepest Function strategy in Express Bug-16.

Figure 4.19 presents the report generated by the First Deepest Function (FDF) strategy for identifying the root cause of the regression. It reports the 'app.enable' function in 'lib/application.js' as the root cause of the regression. As with previous cases, the FDF strategy fails in the Express Bug-16 case due to noisy functions and bad matching, leading to an incorrect identification of the root cause.

```
Function (Counts)
---------------------------------
lib/application.js#FuncExpr@app.set (12)
lib/router/index.js#FuncExpr@proto.match_layer (3)
lib/router/index.js#FuncExpr@proto.process_params (3)
lib/router/layer.js#FuncExpr@Layer.prototype.handle_request (3)
```

Fig. 4.20: Report of Top-n strategy in Express Bug-16.

Figure 4.20 shows the output of the Top-n strategy in the Express Bug-16 case, listing the functions most frequently appearing in the execution trace differences between the base and buggy versions. Similar to Express Bug-1 case, the Top-n strategy also fails to identify the real cause due to the big refactoring where function which introduces the regression has not even executed in the base version.

4.2.6 Express Bug-18

```
1  var express = require('../')
2    , request = require('supertest');
3
4  describe('app', function(){
5    describe('.param(name, fn)', function(){
6      it('should call when values differ when using "next"',
           function(done) {
7        var app = express();
8        var called = 0;
9        var count = 0;
10
11       app.param('user', function(req, res, next, user) {
12         called++;
13         if (user === 'foo') return next('route');
14         req.user = user;
15         next();
16       });
17
18       app.get('/:user/bob', function(req, res, next) {
19         count++;
20         next();
21       });
22       app.get('/foo/:user', function(req, res, next) {
23         count++;
24         next();
25       });
26       app.use(function(req, res) {
27         res.end([count, called, req.user].join(' '));
28       });
29
30       request(app)
31       .get('/foo/bob')
32       .expect('1 2 bob', done);
33     })
34   })
35 })
```

Fig. 4.21: Test code to be executed in Express Bug-18.

Figure 4.21 shows the test code executed for Express Bug-18. This test is intended to check the behavior of the Express app's param function when parameter values differ, using the next callback. The test sets up an Express app that defines a param middleware for the user parameter, and routes to handle requests involving this parameter.

```
0 passing (7s)
1 failing

1) app .param(name, fn) should call when values differ when using
     "next":
   Error: expected '1 2 bob' response body, got '0 1 '
    at error (/home/icefox99/BugJS/express-result/Bug-18/project/
        express_run/node_modules/supertest/lib/test.js:227:13)
    at Test.assert (/home/icefox99/BugJS/express-result/Bug-18/
        project/express_run/node_modules/supertest/lib/test.js
        :172:21)
    at /home/icefox99/BugJS/express-result/Bug-18/project/
        express_run/node_modules/supertest/lib/test.js:125:10
    at Request.callback (/home/icefox99/BugJS/express-result/Bug
        -18/project/express_run/node_modules/superagent/lib/node/
        index.js:728:30)
    at Test.<anonymous> (/home/icefox99/BugJS/express-result/Bug
        -18/project/express_run/node_modules/superagent/lib/node/
        index.js:135:10)
    at Test.emit (node:events:520:28)
    at IncomingMessage.<anonymous> (/home/icefox99/BugJS/express-
        result/Bug-18/project/express_run/node_modules/superagent
        /lib/node/index.js:886:12)
    at IncomingMessage.emit (node:events:532:35)
    at endReadableNT (node:internal/streams/readable:1696:12)
    at process.processTicksAndRejections (node:internal/process/
        task_queues:82:21)
```

Fig. 4.22: Output of regression bug in Express Bug-18.

Figure 4.22 displays the shell output of the test code above in the Express Bug-18 regression commit. The test fails because the response body does not match the expected '1 2 bob'. Instead, it returns '0 1 ', indicating that the param middleware and subsequent route handling did not function as expected.

```
[TEST 132 & 132] lib/express.js#Func@createApplication/
    FuncVar@app,33a3a6c
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,0df81d5~~~~~~~~~~~~~~~~~~~~
[CODE]
function (setting, val) {
  if (1 == arguments.length) {
    return this.settings[setting];
  } else {
    this.settings[setting] = val;
    if (setting === 'trust proxy') {
      debug('compile trust proxy %j', val);
      this.set('trust proxy fn', compileTrust(val));
    }
    return this;
  }
}

[DETAILS]
index: [132,0,0,0]
caller: lib/application.js#AnonFunc@4c5011c/FuncExpr@app[method],
    d2e1ded
isCodeChanged: false
isBeforeThisChanged: true
isBeforeArgsChanged: false
isAfterThisChanged: true
isAfterArgsChanged: false
isRetChanged: false

[ANALYSIS]
Probably caused by the different arguments, please check its
    CALLER "lib/application.js#AnonFunc@4c5011c/FuncExpr@app[
    method],d2e1ded" and the different arguments that passed to
    this function.

...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,0df81d5~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.23: Report of First Deepest Function strategy in Express Bug-18.

Figure 4.23 presents the report generated by the First Deepest Function (FDF) strategy for identifying the root cause of the regression. It reports the 'FuncExpr@app[method]' function in 'lib/application.js' as the root cause of the regression. As with previous cases, the FDF strategy fails in the Express Bug-18 case due to noisy functions and bad matching, leading to an incorrect identification of the root cause.

```
Function (Counts)
-----------------------------------
lib/router/index.js#FuncExpr@proto.process_params (4)
lib/application.js#FuncExpr@app.set (2)
lib/express.js#Func@createApplication/FuncVar@app (1)
lib/application.js#FuncExpr@app.handle (1)
```

Fig. 4.24: Report of Top-n strategy in Express Bug-18.

Figure 4.24 shows the output of the Top-n strategy in the Express Bug-18 case, listing

the functions most frequently appearing in the execution trace differences between the base and buggy versions. Unlike the FDF strategy, the Top-n strategy successfully includes the root cause 'proto.process_params' function in its list of candidates, again demonstrating its effectiveness in this case.

4.2.7 Express Bug-27

```
var express = require('../')
  , request = require('supertest');

describe('app', function(){
  describe('.param(name, fn)', function(){
    it('should defer all the param routes', function(done){
      var app = express();

      app.param('id', function(req, res, next, val){
      if (val === 'new') return next('route');
        return next();
      });

      app.all('/user/:id', function(req, res){
        res.send('all.id');
      });

      app.get('/user/:id', function(req, res){
        res.send('get.id');
      });

      app.get('/user/new', function(req, res){
        res.send('get.new');
      });

      request(app)
      .get('/user/new')
      .expect('get.new', done);
    })
  })
})
```

Fig. 4.25: Test code to be executed in Express Bug-27.

Figure 4.25 shows the test code executed for Express Bug-27 case. This test aims to check the behavior of the Express app's param function, ensuring it defers all the parameterized routes when a specific condition is met.

```
0 passing (15s)
1 failing

1) app .param(name, fn) should defer all the param routes:
   Error: expected 'get.new' response body, got 'get.id'
    at error (/home/icefox99/BugJS/express-result/Bug-27/project/
        express_run/node_modules/supertest/lib/test.js:227:13)
    at Test.assert (/home/icefox99/BugJS/express-result/Bug-27/
        project/express_run/node_modules/supertest/lib/test.js
        :172:21)
    at /home/icefox99/BugJS/express-result/Bug-27/project/
        express_run/node_modules/supertest/lib/test.js:125:10
    at Request.callback (/home/icefox99/BugJS/express-result/Bug
        -27/project/express_run/node_modules/superagent/lib/node/
        index.js:728:30)
    at Test.<anonymous> (/home/icefox99/BugJS/express-result/Bug
        -27/project/express_run/node_modules/superagent/lib/node/
        index.js:135:10)
    at Test.emit (node:events:520:28)
    at IncomingMessage.<anonymous> (/home/icefox99/BugJS/express-
        result/Bug-27/project/express_run/node_modules/superagent
        /lib/node/index.js:886:12)
    at IncomingMessage.emit (node:events:532:35)
    at endReadableNT (node:internal/streams/readable:1696:12)
    at process.processTicksAndRejections (node:internal/process/
        task_queues:82:21)
```

Fig. 4.26: Output of regression bug in Express Bug-27.

Figure 4.26 displays the shell output of the test code above in the Express Bug-27 regression commit. The test fails because the response body does not match the expected 'get.new'. Instead, it returns 'get.id', indicating that the param middleware and subsequent route handling did not function as expected.

```
[TEST 131 & 131] test/app.param.js#AnonFunc@f1dc271/
    AnonFunc@18393e7/AnonFunc@4239e69,4239e69
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,0df81d5~~~~~~~~~~~~~~~~~~~~
[CODE]
function (setting, val) {
  if (1 == arguments.length) {
    return this.settings[setting];
  } else {
    this.settings[setting] = val;
    if (setting === 'trust proxy') {
      debug('compile trust proxy %j', val);
      this.set('trust proxy fn', compileTrust(val));
    }
    return this;
  }
}

[DETAILS]
index: [131,0,0,0,0,0]
caller: lib/application.js#FuncExpr@app.enable,a98a97d
isCodeChanged: false
isBeforeThisChanged: true
isBeforeArgsChanged: false
isAfterThisChanged: true
isAfterArgsChanged: false
isRetChanged: true

[ANALYSIS]
Probably caused by the different arguments, please check its
    CALLER "lib/application.js#FuncExpr@app.enable,a98a97d" and
    the different arguments that passed to this function.

...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/application.js#FuncExpr@app.set
    ,0df81d5~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.27: Report of First Deepest Function strategy in Express Bug-27.

Figure 4.27 presents the report generated by the First Deepest Function (FDF) strategy for identifying the root cause of the regression. It reports the 'app.enable' function in 'lib/application.js' as the root cause of the regression. Similar to previous cases, the FDF strategy fails in the Express Bug-27 case due to noisy functions and bad matching, leading to an incorrect identification of the root cause.

```
Function (Counts)
---------------------------------
lib/application.js#FuncExpr@app.all/AnonFunc@68e1170 (34)
lib/router/route.js#AnonFunc@cea325e/FuncExpr@Route.prototype[
    method] (34)
lib/application.js#FuncExpr@app.set (13)
lib/router/index.js#FuncExpr@proto.process_params (4)
```

Fig. 4.28: Report of Top-n strategy in Express Bug-27.

Figure 4.28 shows the output of the Top-n strategy in the Express Bug-27 case, listing the functions most frequently appearing in the execution trace differences between the base

and buggy versions. Unlike the FDF strategy, the Top-n strategy successfully includes root cause 'proto.process_params' function in its list of candidates.

4.2.8 ESLint Bug-10

```
...

describe("Config", () => {
  ...
  describe("new Config()", () => {
    it("should create config object when using baseConfig with extends", () => {
      const customBaseConfig = {
        extends: path.resolve(__dirname, "..", "fixtures", "config-extends", "array", ".eslintrc")
      };
      const configHelper = new Config({ baseConfig: customBaseConfig }, linter);

      assert.deepEqual(configHelper.baseConfig.env, {
        browser: false,
        es6: true,
        node: true
      });
      assert.deepEqual(configHelper.baseConfig.rules, {
        "no-empty": 1,
        "comma-dangle": 2,
        "no-console": 2
      });
    });
  });
});
```

Fig. 4.29: Test code to be executed in ESLint Bug-10.

Figure 4.29 shows the test code executed for ESLint Bug-10. This test is designed to validate the creation of a configuration object when using baseConfig with extends in ESLint.

```
Config
  new Config()
    1) should create config object when using baseConfig with
        extends

0 passing (13ms)
1 failing

1) Config new Config() should create config object when using
    baseConfig with extends:
   TypeError: Cannot read properties of undefined (reading '
       getConfig')
   Referenced from: undefined
    at Bugfox_Original_load (lib/config/config-file.js:535:52)
    at load (lib/config/config-file.js:552:49)
    at Bugfox_Original_7fe8e43 (lib/config/config-file.js:387:34)
    at lib/config/config-file.js:398:113
    at Array.reduceRight (<anonymous>)
    at Bugfox_Original_applyExtends (lib/config/config-file.js
        :378:28)
    at applyExtends (lib/config/config-file.js:411:57)
    at Object.Bugfox_Original_loadObject (lib/config/config-file.
        js:519:35)
    at Object.loadObject (lib/config/config-file.js:526:55)
    at new Config (lib/config.js:36:75)
    at Object.Bugfox_Original_e0b53fd (tests/lib/config.js
        :179:34)
    at Context.<anonymous> (tests/lib/config.js:198:117)
    at process.processImmediate (node:internal/timers:478:21)
```

Fig. 4.30: Output of regression bug in ESLint Bug-10.

Figure 4.30 shows the output of the test run for ESLint Bug-10. The test fails because of a TypeError, indicating an issue with reading 'getConfig' properties from an undefined value. This suggests that there may be a problem with the Config class's handling of its internal configuration loading and creation.

```
[TEST 63 & 63] tests/lib/config.js#AnonFunc@654adbe/
    AnonFunc@3ce9193/AnonFunc@e0b53fd,e0b53fd
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/config.js#Class@Config/
    Method@constructor,3bee095~~~~~~~~~~~~~~~~~~~~
[CODE_DIFF]
@@ -2,24 +2,42 @@
...
+  this.cliConfig = {};
+  Object.keys(cliConfigOptions).forEach(configKey => {
+    const value = cliConfigOptions[configKey];
+    if (value) {
+      this.cliConfig[configKey] = value;
+    }
+  });
 }

[DETAILS]
index: [63,0]
caller: tests/lib/config.js#AnonFunc@654adbe/AnonFunc@3ce9193/
    AnonFunc@e0b53fd,e0b53fd
isCodeChanged: true
isBeforeThisChanged: false
isBeforeArgsChanged: false
isAfterThisChanged: true
isAfterArgsChanged: true
isRetChanged: true

[ANALYSIS]
Probably caused by the update of this function, please check this
     function's modification and its callee.

...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/config.js#Class@Config/
    Method@constructor,3bee095~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.31: Report of First Deepest Function strategy in ESLint Bug-10.

Figure 4.31 shows the results from the First Deepest Function (FDF) strategy applied to ESLint Bug-10. It precisely identifies the update of constructor method of the Config class in 'lib/config.js' as the root cause of the regression, successfully analyzing and resolving this regression.

```
Function (Counts)
-----------------------------------
lib/config.js#Class@Config/Method@constructor (1)
lib/config/plugins.js#Class@Plugins/Method@constructor (1)
```

Fig. 4.32: Report of Top-n strategy in ESLint Bug-10.

Figure 4.32 displays the output of the Top-n strategy for ESLint Bug-10. This strategy also identifies relevant functions, including the 'constructor' method of the 'Config' class. Both the FDF and Top-n strategies successfully pinpoint the root cause of the regression in this case, demonstrating their effectiveness in identifying issues. We notice that both strategies work well when two versions of call graph of one program execution is concise and easy to match.

4.2.9 ESLint Bug-134

```
"use strict";

const rule = require("../../../lib/rules/no-useless-escape"),
  RuleTester = require("../../../lib/testers/rule-tester");

const ruleTester = new RuleTester();

ruleTester.run("no-useless-escape", rule, {
  valid:[
    {code: "var foo = String.raw`\\.`", parserOptions: {
        ecmaVersion: 6}},
    {code: "var foo = myFunc`\\.`", parserOptions: {ecmaVersion:
        6}}
  ],
  invalid: []
});
```

Fig. 4.33: Test code to be executed in ESLint Bug-134.

Figure 4.33 shows the test code executed for ESLint Bug-134. This test aims to check the rule no-useless-escape in ESLint, ensuring it correctly identifies unnecessary escape sequences in template literals.

```
no-useless-escape
  valid
    1) var foo = String.raw`\.`
    2) var foo = myFunc`\.`

0 passing (2s)
2 failing

1) no-useless-escape valid var foo = String.raw`\.`:

  AssertionError [ERR_ASSERTION]: Should have no errors but had
      1: [
    ...
  ]

    + expected - actual

    -1
    +0

    at Bugfox_Original_testValidTemplate (lib/testers/rule-tester
        .js:415:18)
    at testValidTemplate (lib/testers/rule-tester.js:423:68)
    at Context.Bugfox_Original_848fa9a (lib/testers/rule-tester.
        js:492:23)
    at Context.<anonymous> (lib/testers/rule-tester.js:499:127)
    at process.processImmediate (node:internal/timers:478:21)

2) no-useless-escape valid var foo = myFunc`\.`:

    AssertionError [ERR_ASSERTION]: Should have no errors but had
        1: [
    ...
  ]

    + expected - actual

    -1
    +0

    at Bugfox_Original_testValidTemplate (lib/testers/rule-tester
        .js:415:18)
    at testValidTemplate (lib/testers/rule-tester.js:423:68)
    at Context.Bugfox_Original_848fa9a (lib/testers/rule-tester.
        js:492:23)
    at Context.<anonymous> (lib/testers/rule-tester.js:499:127)
    at process.processImmediate (node:internal/timers:478:21)
```

Fig. 4.34: Output of regression bug in ESLint Bug-134.

Figure 4.34 shows the output of the test run for ESLint Bug-134. The test fails because of AssertionError, indicating that the rule no-useless-escape incorrectly flags valid escape sequences in template literals as errors. This suggests an issue with the rule's logic for handling template literals.

```
[TEST 31 & 31] lib/testers/rule-tester.js#PropFunc@run/
    AnonFunc@fbe9ceb/AnonFunc@860ed11/AnonFunc@4cd2192/
    AnonFunc@848fa9a,848fa9a
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/rules/no-useless-escape.js#
    PropFunc@create,0204c84~~~~~~~~~~~~~~~~~~~~
[CODE_DIFF]
@@ -1,25 +1,39 @@
 function (context) {
-  function validate(escapes, node, elm) {
-    const escapeNotFound = escapes.indexOf(elm[0][1]) === -1;
-    const isQuoteEscape = elm[0][1] === node.raw[0];
-    if (escapeNotFound && !isQuoteEscape) {
+  function validate(escapes, node, match) {
+    const isTemplateElement = node.type === "TemplateElement";
+    const escapedChar = match[0][1];
+    let isUnnecessaryEscape = escapes.indexOf(escapedChar) ===
    -1;
+    let isQuoteEscape;

   ...

   return {
-    Literal: check
+    Literal: check,
+    TemplateElement: check
   };
 }


[DETAILS]
index: [31,0,0,4,8,4,3]
caller: lib/testers/rule-tester.js#PropFunc@run/
    Func@runRuleForItem/FuncExpr@rules.get/PropFunc@create,14667
    c0
isCodeChanged: true
isBeforeThisChanged: false
isBeforeArgsChanged: false
isAfterThisChanged: false
isAfterArgsChanged: false
isRetChanged: true

[ANALYSIS]
Probably caused by the update of this function, please check this
     function's modification and its callee.

...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/rules/no-useless-escape.js#
    PropFunc@create,0204c84~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.35: Report of First Deepest Function strategy in ESLint Bug-134.

Figure 4.35 shows the results from the First Deepest Function (FDF) strategy applied to ESLint Bug-134. In the analysis section, it reports the modification of 'create' method in 'lib/rules/no-useless-escape.js' as the root cause of the regression, which meet the expected result.

```
Function (Counts)
---------------------------------
lib/util/comment-event-generator.js#Func@emitComments (40)
lib/util/comment-event-generator.js#Func@emitCommentsExit (20)
lib/util/comment-event-generator.js#Func@emitCommentsEnter (20)
lib/code-path-analysis/code-path-analyzer.js#
    Func@forwardCurrentToHead (18)
```

Fig. 4.36: Report of Top-n strategy in ESLint Bug-134.

Figure 4.36 displays the output of the Top-n strategy for ESLint Bug-134. This strategy is also affected by plentiful noisy function calls in this case, which override the ground truth.

4.2.10 ESLint Bug-307

```
1  "use strict";
2
3  const rule = require("../../../lib/rules/no-multi-spaces"),
4    RuleTester = require("../../../lib/testers/rule-tester");
5
6  const ruleTester = new RuleTester();
7
8  ruleTester.run("no-multi-spaces", rule, {
9    valid: [
10     "foo\t\t+bar"
11   ],
12
13   invalid: []
14 });
```

Fig. 4.37: Test code to be executed in ESLint Bug-307.

Figure 4.37 shows the test code executed for ESLint Bug-307. This test checks the rule no-multi-spaces in ESLint, ensuring it correctly identifies and handles multiple spaces in the code.

```
no-multi-spaces
  valid
    1) foo    +bar

0 passing (52ms)
1 failing

1) no-multi-spaces valid foo    +bar:
   ReferenceError: Cannot access 'result' before initialization
    at Traverser.Bugfox_Original_enter (lib/util/source-code.js
        :229:20)
    at Traverser.enter (lib/util/source-code.js:240:56)
    at Traverser.__execute (node_modules/estraverse/estraverse.js
        :330:31)
    at Traverser.traverse (node_modules/estraverse/estraverse.js
        :434:28)
    ...
    at runRuleForItem (lib/testers/rule-tester.js:297:63)
    at Bugfox_Original_testValidTemplate (lib/testers/rule-tester
        .js:322:24)
    at testValidTemplate (lib/testers/rule-tester.js:332:66)
    at RuleTester.Bugfox_Original_41dd692 (lib/testers/rule-
        tester.js:450:21)
    at Context.<anonymous> (lib/testers/rule-tester.js:457:125)
    at process.processImmediate (node:internal/timers:478:21)
```

Fig. 4.38: Output of regression bug in ESLint Bug-307.

Figure 4.38 shows the output of the test run for ESLint Bug-307. The test fails due to a ReferenceError, indicating that the variable result is being accessed before it is initialized. This suggests a problem within the no-multi-spaces rule logic.

```
[TEST 61 & 61] lib/testers/rule-tester.js#Class@RuleTester/
    Method@run/AnonFunc@51b6cbc/AnonFunc@c25a022/AnonFunc@dd0e62e
    /AnonFunc@41dd692,41dd692
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/rules/no-multi-spaces.js#
    PropFunc@create/PropFunc@Program,91f0088~~~~~~~~~~~~~~~~~~~~
[CODE_DIFF]
@@ -1,39 +1,37 @@
 function () {
-  const source = sourceCode.getText(), allComments = sourceCode.
    getAllComments(), pattern = /[^\s].*? {2,}/g;
-  let parent;
-  while (pattern.test(source)) {
-    if (!isIndexInComment(pattern.lastIndex, allComments)) {
-      const token = sourceCode.getTokenByRangeStart(pattern.
    lastIndex, {
-        includeComments: true
-      });

    ...

+  sourceCode.tokensAndComments.forEach((leftToken, leftIndex,
    tokensAndComments) => {
+    if (leftIndex === tokensAndComments.length - 1) {
+      return;
+    }
+    const rightToken = tokensAndComments[leftIndex + 1];
+    if (leftToken.range[1] + 2 > rightToken.range[0] ||
    leftToken.loc.end.line < rightToken.loc.start.line) {
+      return;
+    }

      ...

+  });
 }


[DETAILS]
index: [61,1,0,5,12,0,0,1,0,0,1]
caller: lib/util/node-event-generator.js#Class@NodeEventGenerator
    /Method@applySelector,04754ee
isCodeChanged: true
isBeforeThisChanged: false
isBeforeArgsChanged: false
isAfterThisChanged: true
isAfterArgsChanged: true
isRetChanged: false

[ANALYSIS]
Probably caused by the update of this function, please check this
     function's modification and its callee.

...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/rules/no-multi-spaces.js#
    PropFunc@create/PropFunc@Program,91f0088~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.39: Report of First Deepest Function strategy in ESLint Bug-307.

Figure 4.39 shows the results from the First Deepest Function (FDF) strategy applied to ESLint Bug-307. It precisely reports the 'Program' method in 'lib/rules/no-multi-

spaces.js' as the root cause of the regression, which meets the ground truth.

```
Function (Counts)
-----------------------------------
lib/testers/rule-tester.js#Class@RuleTester/Method@run/
    Func@testValidTemplate (1)
lib/testers/rule-tester.js#Class@RuleTester/Method@run/
    Func@runRuleForItem (1)
lib/linter.js#Class@Linter/Method@verify (1)
lib/util/traverser.js#Class@Traverser/Method@traverse (1)
```

Fig. 4.40: Report of Top-n strategy in ESLint Bug-307.

Figure 4.40 displays the output of the Top-n strategy for ESLint Bug-307. Due to noisy function calls and few samples in the extracted differences, the expected result is not reported by Top-n strategy.

4.2.11 Hessian.js Bug-2

```
1  "use strict";
2
3  var should = require('should');
4  var hessian = require('../');
5  var utils = require('./utils');
6
7  describe('map.test.js', function () {
8    it('should decode successful when key is null', function () {
9      var data = new Buffer([77, 116, 0, 0, 78, 83, 0, 4, 110, 117,
           108, 108, 122]);
10     var rv = hessian.decode(data);
11     rv.should.eql({null: 'null'});
12   });
13 });
```

Fig. 4.41: Test code to be executed in Hessian.js Bug-2.

Figure 4.41 shows the test code executed for Hessian.js Bug-2. This test aims to decode a buffer when the key is null and check if the output matches the expected result.

```
map.test.js
  1) should decode successful when key is null

0 passing (5ms)
1 failing

1) map.test.js
     should decode successful when key is null:

    AssertionError: expected Object {} to equal Object { null: '
       null' }
    + expected - actual

    -{}
    +{
    +  "null": "null"
    +}

    at Assertion.fail (node_modules/should/cjs/should.js:326:17)
    at Assertion.value (node_modules/should/cjs/should.js:398:19)
    at Context.Bugfox_Original_4ec982c (test/map.test.js:12:19)
    at Context.<anonymous> (test/map.test.js:21:113)
    at process.processImmediate (node:internal/timers:478:21)
```

Fig. 4.42: Output of regression bug in Hessian.js Bug-2.

Figure 4.42 shows the output of the test run for Hessian.js Bug-2. The test fails with an AssertionError, indicating that the decoded object does not match the expected result. The expected object has a key null with the value 'null', but the actual object is empty.

```
[TEST 51 & 51] test/map.test.js#AnonFunc@2147632/AnonFunc@4ec982c
    ,4ec982c
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/v1/decoder.js#FuncExpr@proto.
    _addRef,f212d01~~~~~~~~~~~~~~~~~~~~
[CODE]
function (obj) {
  this.refMap[this.refId++] = obj;
}

[DETAILS]
index: [51,0,1,0,2]
caller: lib/v1/decoder.js#FuncExpr@proto.readObject,025a099
isCodeChanged: false
isBeforeThisChanged: false
isBeforeArgsChanged: true
isAfterThisChanged: true
isAfterArgsChanged: true
isRetChanged: false

[ANALYSIS]
Probably caused by the different arguments, please check its
    CALLER "lib/v1/decoder.js#FuncExpr@proto.readObject,025a099"
    and the different arguments that passed to this function.

...

[COMPARISON]
...
BASE->beforeArgs: [{"$class":"java.util.HashMap","$":{}}]
NEW-->beforeArgs: [{"$class":"java.util.HashMap","$":{"$map"
    :{}}}]
DIFF:
@@ -1,6 +1,8 @@
 [
   {
     "$class": "java.util.HashMap",
-    "$": {}
+    "$": {
+      "$map": {}
+    }
   }
 ]
...
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/v1/decoder.js#FuncExpr@proto.
    _addRef,f212d01~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.43: Report of First Deepest Function strategy in Hessian.js Bug-2.

Figure 4.43 shows the results from the First Deepest Function (FDF) strategy applied to Hessian.js Bug-2. The strategy successfully identifies the function 'proto.readObject' in 'lib/v1/decoder.js' as root cause of the issue. The analysis suggests that the different arguments passed to function 'proto._addRef' in 'lib/v1/decoder.js' might be causing the problem. The differences of function arguments between two version explicitly pointed out the issue might be caused by the update of code dealing with hash map in that commit.

```
Function (Counts)
---------------------------------
lib/v1/decoder.js#FuncExpr@proto.read (3)
index.js#FuncExpr@exports.decode (1)
lib/v1/decoder.js#FuncExpr@proto.read (1)
lib/v1/decoder.js#FuncExpr@proto.readObject (1)
```

Fig. 4.44: Report of Top-n strategy in Hessian.js Bug-2.

Figure 4.44 displays the output of the Top-n strategy for Hessian.js Bug-2. This strategy identifies functions based on their invocation counts. Both the FDF and Top-n strategies successfully point to the 'proto.readObject' function in 'lib/v1/decoder.js' as the source of the regression.

4.2.12 Hessian.js Bug-8

```
'use strict';
...

describe('hessian v1', function () {
  afterEach(function () {
    encoder.clean();
    decoder.clean();
  });

    describe('long', function () {
    it('should write and read long ok', function () {
      var tests = [
      ...
        ['9007199254740992', '<Buffer 4c 00 20 00 00 00 00 00 00>
            '],
        ['9007199254740993', '<Buffer 4c 00 20 00 00 00 00 00 01>
            '],
        ['9223372036854775807', '<Buffer 4c 7f ff ff ff ff ff ff
            ff>'],
      ];

      tests.forEach(function (t) {
        var buf = encoder.writeLong(t[0]).get();
        buf.inspect().should.equal(t[1]);
        decoder.init(buf).readLong().should.eql(t[0]);
        encoder.clean();
        decoder.clean();
      });
    });
  });
});
```

Fig. 4.45: Test code to be executed in Hessian.js Bug-8.

Figure 4.45 shows the test code executed for Hessian.js Bug-8. This test aims to write and read various long integer values and check if the output matches the expected buffer

representation.

```
hessian v1
  long
    1) should write and read long ok

0 passing (30ms)
1 failing

1) hessian v1
     long
       should write and read long ok:
   AssertionError: expected 9007199254740992 to equal '
      9007199254740992'
    at Assertion.fail (node_modules/should/lib/assertion.js
       :196:17)
    at prop.<computed> (node_modules/should/lib/assertion.js
       :81:17)
    at Bugfox_Original_efe399a (test/v1.test.js:41:53)
    at /home/icefox99/BugJS/hessian-result/Bug-8/project/
       hessian_run/test/v1.test.js:50:121
    at Array.forEach (<anonymous>)
    at Context.Bugfox_Original_fa86bb7 (test/v1.test.js:37:19)
    at Context.<anonymous> (test/v1.test.js:62:117)
    at process.processImmediate (node:internal/timers:478:21)
```

Fig. 4.46: Output of regression bug in Hessian.js Bug-8.

Figure 4.46 shows the output of the test run for Hessian.js Bug-8. The test fails with an AssertionError, indicating that the decoded value does not match the expected result. The expected value is a string '9007199254740992', but the actual value is a number 9007199254740992. It is clearly a type error when handling long data type in JavaScript.

```
[TEST 53 & 53] test/v1.test.js#AnonFunc@52d7217/AnonFunc@3df64bc/
    AnonFunc@fa86bb7,fa86bb7
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/utils.js#FuncExpr@exports.
    handleLong,bc708aa~~~~~~~~~~~~~~~~~~~~
[CODE_DIFF]
@@ -1,8 +1,8 @@
 function (val) {
-  if (val.greaterThan(MAX_SAFE_INT) || val.lessThan(MIN_SAFE_INT
    )) {
-    val = val.toString();
-    debug('[hessian.js Warning] Read a not safe long(%s),
    translate it to string', val);
-    return val;
+  var notSafeInt = val.high > MAX_INT_HIGH || val.high ===
    MAX_INT_HIGH && val.low > 0 || val.high < -1 * MAX_INT_HIGH
    || val.high === -1 * MAX_INT_HIGH && val.low < 0;
+  if (notSafeInt) {
+    debug('[hessian.js Warning] Read a not safe long, translate
    it to string');
+    return val.toString();
   }
   return val.toNumber();
 }


[DETAILS]
index: [53,6,3,1]
caller: lib/v1/decoder.js#FuncExpr@proto.readLong,29bd6be
isCodeChanged: true
isBeforeThisChanged: false
isBeforeArgsChanged: false
isAfterThisChanged: false
isAfterArgsChanged: false
isRetChanged: true

[ANALYSIS]
Probably caused by the update of this function, please check this
     function's modification and its callee.

...

[COMPARISON]
...
BASE->returnVal: "9007199254740992"
NEW-->returnVal: 9007199254740992
DIFF:
@@ -1 +1 @@
-"9007199254740992"
+9007199254740992
~~~~~~~~~~~~~~~~~~~~FUNCTION: lib/utils.js#FuncExpr@exports.
    handleLong,bc708aa~~~~~~~~~~~~~~~~~~~~
```

Fig. 4.47: Report of First Deepest Function strategy in Hessian.js Bug-8.

Figure 4.47 shows the results from the First Deepest Function (FDF) strategy applied to Hessian.js Bug-8. The strategy successfully identifies the function 'exports.handleLong' in 'lib/utils.js' as the root cause of the issue. The analysis suggests that the update of this function might be causing the problem due to changes in how it handles the long integer values.

```
Function (Counts)
---------------------------------
test/v1.test.js#AnonFunc@52d7217/AnonFunc@3df64bc/
    AnonFunc@fa86bb7/AnonFunc@efe399a (1)
lib/v1/decoder.js#FuncExpr@proto.readLong (1)
lib/utils.js#FuncExpr@exports.handleLong (1)
lib/v1/decoder.js#FuncExpr@proto.handleType (1)
```

Fig. 4.48: Report of Top-n strategy in Hessian.js Bug-8.

Figure 4.48 displays the output of the Top-n strategy for Hessian.js Bug-8. This strategy identifies functions based on their invocation counts. Both the FDF and Top-n strategies successfully point to the 'exports.handleLong' function in 'lib/utils.js' as the source of the regression.

4.3 RQ2: Can Bugfox meet the performance demands in real-world development?

To evaluate the performance of our system, we record the running time of different stages in Bugfox, the lines of code (LOC) of each test project, and the size of tracing log of entire program. The size of tracing log is used as the approximation of memory overhead.

In terms of code transformation, Table 4.3 demonstrates the lines of code (LoC) of each regression and their corresponding transformation time. It can be seen that the regressions from ESLint project spend more time that other regressions. This is because that ESLint, as a static code analysis tool for identifying problematic patterns find in JavaScript code, contains a huge numbers of unit tests, resulting in thousands of functions being transformed in runtime. However, since code transformation is a one-time task in our workflow and all such transformations spend only a few seconds, we argue that the time of code transformation is acceptable in real usage of this system.

Project	Bug-ID	Lines of code	Code transformation runtime
Express	1	7211	1.145 s
Express	8	10189	1.426 s
Express	9	9631	1.393 s
Express	13	7630	1.153 s
Express	16	9653	1.403 s
Express	18	8309	1.231 s
Express	27	8306	1.206 s
ESLint	10	222470	7.331 s
ESLint	134	169489	6.422 s
ESLint	307	225086	7.335 s
hessian.js	2	4805	1.016 s
hessian.js	8	4339	950.017 ms

Table 4.3: Lines of code and code transformation runtime.

Table 4.4 demonstrates the overall performance of Bugfox on memory and runtime, aiming to evaluate the overhead of applying tracing in our system. The 3rd column and 4th column show the runtime of the original program and instrumented program. The comparison of these two columns illustrates the runtime overhead of tracing. It can be seen that the overhead varies widely in our experiment. For example, in Bug-2 of hessian.js, the runtime of the instrumented program is only 1.3 times larger than the original program, while in Bug-8 of Express, the ratio increases to 2400 times. After inspecting the detail of traces, we speculate that this discrepancy is caused by different amount of data being traced and time spent on their object serialization. However, the maximum runtime of instrumented program in our experiment takes less than 26 seconds, which could be considered acceptable in real-world development process. The 5th column shows the time spent on analyzing differences between traces, and in all cases, the maximum analysis time is less than 10 seconds. The last column demonstrates the size of traces stored in the disk, as an indicator to evaluate the memory overhead of tracing, it varies from 7MB to 132MB. In our experiment, our system spends less than 1 minute for solving a regression with memory overhead less than 200MB for all cases. Compared with manual debugging, Bugfox observably saves time for our users and meet our expected performance demands.

Project	Bug-ID	Original runtime	Runtime of instrumented program	Analysis time of differences	Size of trace
Express	1	12 ms	3 s	527.086 ms	28 MB
Express	8	12 ms	18s	277.583 ms	125MB
Express	9	2 ms	3 s	1.807 s	41 MB
Express	13	12 ms	18 s	4.591 s	93 MB
Express	16	11 ms	26 s	6.457 s	132 MB
Express	18	10 ms	15 s	1.633 s	80 MB
Express	27	13 ms	23 s	8.085 s	99 MB
ESLint	10	24 ms	35 ms	162.41 ms	24 MB
ESLint	134	15 ms	2 s	1.136s	7 MB
ESLint	307	18 ms	236 ms	265.122 ms	8 MB
hessian.js	2	3 ms	4 ms	197.577 ms	39 MB
hessian.js	8	5 ms	33 ms	165.294 ms	41 MB

Table 4.4: Overall performance of Bugfox on memory and runtime.

4.4 Threats to validity

There are several threats to internal validity of our evaluation. The number of regression cases used in our experiment is very limited, which may lead to bias of experimental results. Further benchmarking with different test frameworks is needed to comprehensively evaluate our system. Moreover, the use of heuristic strategies in localizing the suspicious functions also reduces the generalizability of Bugfox. In particular, the selection of value n (number of candidates) in Top-n strategy is highly ad-hoc. Further research is required to analyze the setting of this parameter.

As explained in the failures of Bug-1 and Bug-16 in Express, our current approach

has limitations on dealing with regressions with large scale refactoring or noisy functions, as they are lead to great difficulty on matching two call graphs and extracting usable differences for later analysis. Furthermore, as shown in Table 4.4, the overhead of tracing varies a lot across different regression cases. Although Bugfox can work smoothly on tested modules with small granularity, the overhead of complete tracing would be enormous on middle-level tested modules or integration testing.

Chapter 5

Conclusion

5.1 Summary

Addressing software regression heavily relies on manual debugging in software development, and there is a clear need for automated tools to enhance the efficiency of regression testing. In this thesis, we propose Bugfox as a tool aimed at improving this process. Bugfox operates by inserting instrumentation into programs to trace their complete execution, allowing for the localization of regression causes by comparing the execution traces of both the clean and faulty versions of a program.

In our evaluation, Bugfox was tested on 12 real-world regression cases. The FDF strategy successfully resolved 6 regressions, while the Top-n strategy identified an additional 4 regressions, leading to an overall accuracy of 83%. Remarkably, Bugfox achieves this level of accuracy while maintaining performance, processing each regression in under one minute and using less than 200 megabytes of memory. These results indicate that Bugfox can effectively assist developers in diagnosing and resolving regressions within real-world development environments.

Bugfox represents a practical approach to automating the localization of regression causes, addressing a significant challenge in software maintenance. By utilizing execution traces and heuristic analysis, Bugfox aids developers in identifying the root causes of regressions more efficiently compared to traditional manual debugging methods. Its minimal resource requirements make it feasible for integration into existing development workflows with little additional overhead. Although Bugfox may not be a groundbreaking tool, it demonstrates potential for enhancing the regression testing of JavaScript applications by offering timely and actionable insights into regression bugs.

The tool's ability to automate and streamline the regression debugging process marks a step forward in improving software quality assurance. By reducing the time and effort required for regression localization, Bugfox helps developers maintain the stability and performance of their applications as they evolve.

5.2 Future Work

Our future work is to investigate how to automatically identify and collect the regressions in open-source projects and retest our tool on a more comprehensive benchmark. Another future work is reducing overheads due to obtaining execution trace in JavaScript.

Additionally, we aim to refine the heuristic strategies used by Bugfox to enhance its accuracy and robustness further. Exploring machine learning techniques could be a potential direction for improving regression localization. We also plan to investigate the integration of Bugfox with other automated testing and debugging tools to create a more comprehensive solution for software quality assurance. Finally, extending the application

of Bugfox to other programming languages and environments could broaden its utility and impact, making it a versatile tool for regression testing across different software ecosystems. Through these efforts, we aim to improve Bugfox continuously and ensure its relevance and effectiveness in the evolving landscape of software development.

Publications and Research Activities

(1) Yuefeng Hu, Tetsuro Yamazaki, Shigeru Chiba. Pinpoint the Cause of Software Regression in JavaScript. *PPL 2024*, March, 2024. (Poster)

References

[1] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101, 2019.

[2] Kent Beck, Erich Gamma, David Saff, and Kris Vasudevan. JUnit - a programmer-friendly testing framework for java and the jvm. `https://junit.org/junit5/`, 2024.

[3] Charlie Poole, James Newkirk, Alexei Vorontsov, Michael Two, Philip Craig, Rob Prouse, Simone Busoli, and Neil Colvin. NUnit - a unit-testing framework for all .net languages. `https://nunit.org/`, 2024.

[4] OpenJS Foundation. Mocha - the fun, simple, flexible test framework. `https://mochajs.org/`, 2024.

[5] James Quick. How to debug node.js code in visual studio code. Available at: `https://assets.digitalocean.com/articles/how-to-debug-node-js-code-in-visual-studio-code/6.png`, 2020. Accessed: 2024-07-16.

[6] Kayce Basques, Sofia Emelianova. Debug javascript. Available at: `https://developer.chrome.com/static/docs/devtools/javascript/image/the-scope-pane-2c6ae38b4b0e6_1920.png`, 2024. Accessed: 2024-07-16.

[7] Mike Diaz. Chrome devtools performance tab. Available at: `https://miro.medium.com/v2/resize:fit:1400/format:webp/1*WfFqMJJbb3qY9J6s69uWNA.png`, 2021. Accessed: 2024-07-16.

[8] Fabrizio Pastore, Leonardo Mariani, and Alberto Goffi. Radar: A tool for debugging regression problems in c/c++ software. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1335–1338, 2013.

[9] Djordje Maksimovic, Andreas Veneris, and Zissis Poulos. Clustering-based revision debug in regression verification. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 32–37, 2015.

[10] Mark VanderVoord, Doctor Surly, et al. unity - simple unit testing for c. `https://www.throwtheswitch.org/unity`, 2024.

[11] Cypress.io Inc. cypress - fast, easy and reliable testing for anything that runs in a browser. `https://www.cypress.io/`, 2024.

[12] Pekka Klärck, Janne Härkönen, et al. Robot - generic automation framework for acceptance testing and rpa. `https://robotframework.org/`, 2024.

[13] James Newkirk and Brad Wilson. xUnit - a free, open source, community-focused unit testing tool for .net. `https://xunit.net/`, 2024.

[14] Pivotal Labs. jasmine - simple javascript testing framework for browsers and node.js. `https://jasmine.github.io/`, 2024.

[15] Developer Express Inc. TestCafe - a node.js tool to automate end-to-end web testing. `https://testcafe.io/`, 2024.

[16] Christoph Nakazawa, Orta Therox, Simen Bekkhus, and Ricky. Jest - a delightful javascript testing framework with a focus on simplicity. `https://jestjs.io/`, 2024.

[17] Raúl H. Rosero, Omar S. Gómez, and Glen Rodríguez. 15 years of software regression

testing techniques — a survey. *International Journal of Software Engineering and Knowledge Engineering*, 26(05):675–689, 2016.

[18] Hema Srikanth and Myra B. Cohen. Regression testing in software as a service: An industrial case study. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 372–381, 2011.

[19] Yiyu He Fengmk and Others. hessian.js - a js hessian binary web service protocol. `https://github.com/node-modules/hessian.js`, 2023.

[20] Caucho Technology Inc. Hessian binary web service protocol. `http://hessian.caucho.com/`, 2022.

[21] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, 1990.

[22] Adrian Heine né Lang, Marijn Haverbeke, and Ingvar Stepanyan. acorn - a small, fast, javascript-based javascript parser. `https://github.com/acornjs/acorn`, 2024.

[23] Rich Harris. estree walker - traverse an estree-compliant ast. `https://github.com/Rich-Harris/estree-walker`, 2023.

[24] Hiromu Ishibe. A cause detector of software regressions by comparing program execution traces. Master's thesis, The University of Tokyo, Japan, 2023.

[25] TJ Holowaychuk, StrongLoop, et al. Express - fast, unopinionated, minimalist web framework for node.js. `https://expressjs.com/`, 2024.

[26] Nicholas C. Zakas. ESLint - a tool for identifying and reporting on patterns found in ecmascript/javascript code. `https://eslint.org/`, 2024.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Professor Shigeru Chiba, for his invaluable guidance, support and encouragement throughout the course of this research. I have spent two fulfilling and joyful years in this laboratory.

I would also like to thank Mr. Miyazaki for his advice and guidance on my research, and also for welcoming me to this lab.

I also want to thank Mr. Dai for the help of writing the research paper, and thank Mr. Fu for his advice on implementation of this tool. I also want to thank all my colleagues for their encouragement and advice regarding my master's life, as well as for discussing this research with me.

Lastly, I want to thank my parents for their unwavering support and encouragement throughout my master's life, which gave me the courage to persevere even during difficult times.

A

Experimental Details

Table A.1 shows the detailed experimental results of Bugfox on 12 regression test cases. The 3rd column shows the ground truth of the root cause of regression. The 4th column shows the function reported from FDF strategy and indicates which case it fits in Table 3.1. The 5th column shows top four candidates reported from Top-n strategy with n=4 in descending order of frequency, where the number in parentheses indicates the frequency of each candidate. Reported function in green color indicates that it matches the expected result.

Project	Bug-ID	Expected result	FDF strategy	Top-n strategy with n=4
Express	1	lib/router/index.js# FuncExpr@proto.handle	lib/application.js# FuncExpr@app.enable (Case 5)	lib/application.js#FuncExpr@app.set (4) lib/application.js#AnonFunc@d7d1b0c/FuncExpr@app[method] (4) lib/application.js#FuncExpr@app.enable (2) lib/express.js#Func@createApplication (1)
Express	8	lib/application.js# FuncExpr@app.use	lib/application.js# FuncExpr@app.use (Case 5)	lib/utils.js#FuncExpr@exports.flatten (2) lib/application.js#FuncExpr@app.use (1) lib/application.js#FuncExpr@app.use/AnonFunc@497a64c (1) lib/router/index.js#FuncExpr@proto.use (1)
Express	9	lib/application.js# FuncExpr@app.use	lib/application.js# FuncExpr@app.enable (Case 5)	lib/application.js#FuncExpr@app.set (46) lib/application.js#FuncExpr@app.enabled (6) lib/application.js#FuncExpr@app.use (5) lib/application.js#FuncExpr@app.lazyrouter (5)
Express	13	lib/router/index.js# FuncExpr@proto.process_params	lib/application.js# FuncExpr@app.enable (Case 5)	lib/application.js#FuncExpr@app.set (11) lib/application.js#FuncExpr@app.enabled (3) lib/application.js#AnonFunc@4c5011c/FuncExpr@app[method] (3) lib/router/index.js#FuncExpr@proto.process_params (3)
Express	16	lib/response.js# FuncExpr@res.redirect	lib/application.js# FuncExpr@app.enable (Case 5)	lib/application.js#FuncExpr@app.set (12) lib/router/index.js#FuncExpr@proto.match_layer (3) lib/router/index.js#FuncExpr@proto.process_params (3) lib/router/layer.js#FuncExpr@Layer.prototype.handle_request (3)
Express	18	lib/router/index.js# FuncExpr@proto.process_params	lib/application.js# AnonFunc@4c5011c/FuncExpr@app[method] (Case 5)	lib/router/index.js#FuncExpr@proto.process_params (4) lib/application.js#FuncExpr@app.set (2) lib/express.js#Func@createApplication/FuncVar@app (1) lib/application.js#FuncExpr@app.handle (1)
Express	27	lib/router/index.js# FuncExpr@proto.process_params	lib/application.js# FuncExpr@app.enable (Case 5)	lib/application.js#FuncExpr@app.all/AnonFunc@68e1170 (34) lib/router/route.js#AnonFunc@cea325e/FuncExpr@Route.prototype[method] (34) lib/application.js#FuncExpr@app.set (13) lib/router/index.js#FuncExpr@proto.process_params (4)
ESLint	10	lib/config.js# Class@Config/Method@constructor	lib/config.js# Class@Config/Method@constructor (Case 2)	lib/config.js#Class@Config/Method@constructor (1) lib/config/plugins.js#Class@Plugins/Method@constructor (1)
ESLint	134	lib/rules/no-useless-escape.js# PropFunc@create	lib/rules/no-useless-escape.js# PropFunc@create (Case 2)	lib/util/comment-event-generator.js#Func@emitComments (40) lib/util/comment-event-generator.js#Func@emitCommentsExit (20) lib/util/comment-event-generator.js#Func@emitCommentsEnter (20) lib/code-path-analysis/code-path-analyzer.js#Func@forwardCurrentToHead (18)
ESLint	307	lib/rules/no-multi-spaces.js# PropFunc@create/PropFunc@Program	lib/rules/no-multi-spaces.js# PropFunc@create/PropFunc@Program (Case 2)	lib/testers/rule-tester.js#Class@RuleTester/Method@run/Func@testValidTemplate (1) lib/testers/rule-tester.js#Class@RuleTester/Method@run/Func@runRuleForItem (1) lib/linter.js#Class@Linter/Method@verify (1) lib/util/traverser.js#Class@Traverser/Method@traverse (1)
hessian.js	2	lib/v1/decoder.js# FuncExpr@proto.readObject	lib/v1/decoder.js# FuncExpr@proto.readObject (Case 1)	lib/v1/decoder.js#FuncExpr@proto.read (3) index.js#FuncExpr@exports.decode (1) lib/v1/decoder.js#FuncExpr@proto.read (1) lib/v1/decoder.js#FuncExpr@proto.readObject (1)
hessian.js	8	lib/utils.js# FuncExpr@exports.handleLong	lib/utils.js# FuncExpr@exports.handleLong (Case 2)	test/v1.test.js#AnonFunc@52d7217/AnonFunc@3df64bc/AnonFunc@fa86bb7/AnonFunc@efe399a (1) lib/v1/decoder.js#FuncExpr@proto.readLong (1) lib/utils.js#FuncExpr@exports.handleLong (1) lib/v1/decoder.js#FuncExpr@proto.handleType (1)

Table A.1: Detailed experimental results of Bugfox on 12 regression test cases.