

**DISSERTATION**

SUBMITTED

TO THE

COMBINED FACULTY FOR THE NATURAL SCIENCES AND FOR MATHEMATICS

OF

HEIDELBERG UNIVERSITY, GERMANY

FOR THE DEGREE OF

DOCTOR OF NATURAL SCIENCES

PUT FORWARD BY

**MOHAMMADREZA GHANAVATI (M.Sc.)**

FROM RAMHORMOZ (IRAN)

ORAL EXAMINATION:



# AUTOMATED FAULT LOCALIZATION IN LARGE JAVA APPLICATIONS

ADVISOR: PROF. DR. ARTUR ANDRZEJAK



# Abstract

Modern software systems evolve steadily. Software developers change the software codebase every day to add new features, to improve the performance, or to fix bugs. Despite extensive testing and code inspection processes before releasing a new software version, the chance of introducing new bugs is still high. A code that worked yesterday may not work today, or it can show a degraded performance causing software regression. The laws of software evolution state that the complexity increases as software evolves. Such increasing complexity makes software maintenance harder and more costly. In a typical software organization, the cost of debugging, testing, and verification can easily range from 50% to 75% of the total development costs.

Given that human resources are the main cost factor in the software maintenance and the software codebase evolves continuously, this dissertation tries to answer the following question: How can we help developers to localize the software defects more effectively during software development? We answer this question in three aspects.

First, we propose an approach to localize failure-inducing changes for crashing bugs. Assume the source code of a buggy version, a failing test, the stack trace of the crashing site, and a previous correct version of the application. We leverage program analysis to contrast the behavior of the two software versions under the failing test. The difference set is the code statements which contribute to the failure site with a high probability.

Second, we extend the version comparison technique to detect the leak-inducing defects caused by software changes. Assume two versions of a software codebase (one previous non-leaky and the current leaky version) and the existing test suite of the application. First, we compare the memory footprint of the code locations between two versions. Then, we use a confidence score to rank the suspicious code statements, i.e., those statements which can be the potential root causes of memory leaks. The higher the score, the more likely the code statement is a potential leak.

Third, our observation on the related work about debugging and fault localization reveals that there is no empirical study which characterizes the properties of the leak-inducing defects and their repairs. Understanding the characteristics of the real defects caused by resource and memory leaks can help both researchers and practitioners to

improve the current techniques for leak detection and repair. To fill this gap, we conduct an empirical study on 491 reported resource and memory leak defects from 15 large Java applications. We use our findings to draw implications for leak avoidance, detection, localization, and repair.

# Zusammenfassung

Moderne Softwaresysteme entwickeln sich ständig weiter. Softwareentwickler ändern jeden Tag die Codebasis der Software, um neue Funktionen hinzuzufügen, die Leistung zu verbessern oder Fehler zu beheben. Trotz umfangreicher Test- und Code-Inspektionsprozesse vor der Veröffentlichung einer neuen Softwareversion ist die Chance, neue Fehler einzuführen, immer noch hoch. Ein Code, der gestern funktioniert hat, funktioniert heute möglicherweise nicht, oder er kann eine verminderte Leistung aufweisen, die eine Software-Regression verursacht. Die Gesetze der Softwareentwicklung besagen, dass die Komplexität mit der Entwicklung der Software zunimmt. Diese zunehmende Komplexität macht die Softwarepflege schwieriger und kostspieliger. In einem typischen Softwareunternehmen können die Kosten für Debugging, Test und Verifikation leicht zwischen 50% und 75% des gesamten Entwicklungskosten betragen.

Da die Personalressourcen der Hauptkostenfaktor in der Softwarepflege sind und sich die Software-Codebasis kontinuierlich weiterentwickelt, versucht diese Arbeit, die folgende Frage zu beantworten: Wie können wir Entwicklern helfen, die Fehler während der Softwareentwicklung effektiver zu lokalisieren? Wir beantworten diese Frage in drei Aspekten.

Zuerst schlagen wir einen Ansatz zur Lokalisierung von fehlerinduzierenden Änderungen für abstürzende Fehler vor. Nehmen wir den Quellcode einer fehlerhaften Version, einen fehlerhaften Test, den Stack-Trace der abstürzenden Seite und eine vorherige korrekte Version der Anwendung an. Wir nutzen die Programmanalyse, um das Verhalten der beiden Softwareversionen unter dem Misserfolgstest gegenüberzustellen. Der Differenzsatz sind die Codeanweisungen, die mit hoher Wahrscheinlichkeit zur Fehlerstelle beitragen.

Zweitens erweitern wir die Methode des Versionsvergleichs, um die leckinduzierenden Fehler zu erkennen, die durch Softwareänderungen verursacht werden. Annehmend zwei Versionen einer Software-Codebasis (eine vorhergehende leckfreie und die aktuelle leckbehaftete Version) und die bestehende Testsuite der Anwendung. Zuerst vergleichen wir den Speicherbedarf der Codepositionen

zwischen zwei Versionen. Dann verwenden wir einen Vertrauensscore, um die verdächtigen Code-Anweisungen zu bewerten, d. h. diejenigen Anweisungen, die die potenziellen Grundursachen für Speicherlecks sein können. Je höher der Wert, desto wahrscheinlicher ist es, dass die Code-Anweisung ein potenzielles Leck ist.

Drittens zeigt unsere Beobachtung der damit verbundenen Arbeiten zum Debugging und zur Fehlerlokalisierung, dass es keine empirische Studie gibt, die die Eigenschaften der leckinduzierenden Defekte und ihrer Reparaturen charakterisiert. Das Verständnis der Eigenschaften der realen Defekte, die durch Ressourcen- und Speicherlecks verursacht werden, kann sowohl Forschern als auch Praktikern helfen, die aktuellen Techniken zur Leckerkennung und -behebung zu verbessern. Um diese Lücke zu schließen, führen wir eine empirische Studie über 491 gemeldete Ressourcen- und Speicherleckfehler aus 15 großen Java-Anwendungen durch. Wir nutzen unsere Ergebnisse, um Implikationen für die Vermeidung, Erkennung, Lokalisierung und Reparatur von Lecks zu ermitteln.



## Acknowledgements

First and foremost, I would like to thank my advisor, Prof. Dr. Artur Andrzejak, for his great support and guidance during my Ph.D. study. I learned from him everything I know of research in the field of software debugging. He has taught me how to think about research problems and helped me make significant progress in skills that are essential for a researcher. He has always been supportive of any enterprise I have undertaken. His influence is present on every page of this thesis. I wish that I can use the skills I learned from him in my future research career.

I would like to thank Prof. Dr. Ulrich Brüning, Prof. Dr. Holger Fröning, and Dr. Christoph Garbe for agreeing to serve in my doctoral committee.

I would like to thank Prof. Dr. David Lo from Singapore Management University for our collaborative work on the empirical study of resource and memory leaks that appears in this dissertation. It was always my pleasure to work with David who was willing to contribute generously.

I gratefully acknowledge the support that I received from the Faculty of Mathematics and Computer Science at Heidelberg University and from the members of the Parallel and Distributed Systems group. I thank my dear friends and lab mates, Zhen Dong, Lutz Büch, Diego Costa, and Thomas Bach for the many inspiring discussions on various research topics. I would like to especially thank Diego for our collaborative work on the empirical study of resource and memory leaks which appears in this dissertation.

I deeply thank Heidelberg Graduate School of Mathematical and Computational Methods for the Sciences (HGS MathComp) for providing me the financial support for traveling to Software Engineering conferences during my study.

I am also grateful to Anke Sopka for her support in the administrative work.

Finally, I would like to thank my family: my parents, my wife Nasrin, our two sons, Amir and Amin, and my parents-in-law. They were sources of love in my life. I like to especially thank my dear wife Nasrin, who was a great supporter of me during every day of my study. She was taking care of everything in our daily life, providing a relaxed environment for me to focus on my study. I also like to thank my dear parents

for their excellent support during my life and my education. Without them, I could not reach this point. I honorably dedicate my thesis to my parents and my wife.

To my parents and my wife.



# Contents

Abstract . . . . .	v
Zusammenfassung . . . . .	vii
Acknowledgements . . . . .	ix
List of Tables . . . . .	xvii
List of Figures . . . . .	xxi
<b>1 Introduction</b>	<b>1</b>
1.1 Scalable Isolation of Failure-Inducing Changes . . . . .	2
1.2 Automated Memory Leak Diagnosis via Version Comparison . . . . .	3
1.3 Empirical Study on Resource and Memory Leaks . . . . .	5
1.4 Contributions . . . . .	6
1.5 Papers Appeared . . . . .	7
1.6 Overview and Organization . . . . .	8
<b>2 Background and Related Work</b>	<b>10</b>
2.1 Fault Localization of Functional Bugs . . . . .	10
2.1.1 Program Analysis . . . . .	11
2.1.2 Fault Localization Techniques . . . . .	14
2.2 Resource and Memory Leak Detection in Java . . . . .	18
2.2.1 Memory Management in Java . . . . .	18
2.2.2 Resource and Memory Leaks . . . . .	22
2.2.3 Debugging Leak-Inducing Defects . . . . .	22
<b>3 Scalable Isolation of Failure-Inducing Changes</b>	<b>27</b>
3.1 Introduction . . . . .	27
3.1.1 Core Idea . . . . .	28

3.1.2	Contribution . . . . .	29
3.2	Version Comparison Approach . . . . .	30
3.2.1	Approach Overview . . . . .	31
3.2.2	Discussion . . . . .	32
3.3	Experimental Design . . . . .	33
3.4	Experimental Evaluation . . . . .	34
3.4.1	Case Studies . . . . .	35
3.4.2	Complexity of the Approach . . . . .	40
3.4.3	Performance Evaluation . . . . .	41
3.5	Chapter Summary . . . . .	42
<b>4</b>	<b>Automated Memory Leak Diagnosis via Version Comparison</b>	<b>43</b>
4.1	Introduction . . . . .	44
4.1.1	Core Idea . . . . .	46
4.1.2	Contributions . . . . .	46
4.2	Leak Detection via Version Comparison . . . . .	47
4.2.1	Approach Description . . . . .	48
4.2.2	Instrumentation and Data Collection . . . . .	48
4.2.3	Types of Allocation Sites . . . . .	49
4.2.4	Leak Confidence Score . . . . .	52
4.2.5	Ranking . . . . .	54
4.2.6	Discussion . . . . .	54
4.3	Experimental Design . . . . .	55
4.3.1	Methodology . . . . .	55
4.3.2	Research Questions . . . . .	59
4.4	Experimental Evaluation . . . . .	60
4.4.1	Experiment I: Evaluation of Synthetic Defects . . . . .	60
4.4.2	Answer to RQ2: Evaluation of Real-World Issues . . . . .	62
4.4.3	Answer to RQ3: Analysis of Factors Contributing to LC . . . . .	70
4.4.4	Answer to RQ4: Evaluation of Runtime and Memory Efficiency . . . . .	71
4.5	Discussion . . . . .	72

4.5.1	What is the Distribution of the Leak Confidence Value for Various Software Projects? . . . . .	72
4.5.2	Does Our Approach Help Developers to Detect Memory Leaks? . . . . .	73
4.5.3	Can Our Approach Find the Root Cause of the Memory Leaks? . . . . .	74
4.6	Threats to Validity . . . . .	75
4.7	Chapter Summary . . . . .	75
<b>5</b>	<b>An Empirical Study on Leak-inducing Defects and Their Repairs</b>	<b>77</b>
5.1	Introduction . . . . .	77
5.2	Background . . . . .	81
5.2.1	Issue Report . . . . .	81
5.3	Empirical Study Design . . . . .	82
5.3.1	Studied Projects . . . . .	82
5.3.2	Research Questions . . . . .	85
5.3.3	Data Extraction . . . . .	86
5.3.4	Tagging Leak-Related Defects . . . . .	88
5.3.5	Uniqueness of Categories . . . . .	89
5.4	Empirical Study Results . . . . .	90
5.4.1	RQ1: What Is Distribution of Leak Types in Studied Projects? . . . . .	90
5.4.2	RQ2: How Are Leak-Related Defects Detected? . . . . .	92
5.4.3	RQ3: To What Extent Are the Leak-Inducing Defects Localized? . . . . .	97
5.4.4	RQ4: What Are the Most Common Root Causes? . . . . .	100
5.4.5	RQ5: What Are the Characteristics of the Repair Patches? . . . . .	103
5.4.6	RQ6: How Complex Are Repairs of the Leak-Inducing Defects? . . . . .	111
5.4.7	Other Findings . . . . .	115
5.5	Implications . . . . .	119
5.6	Threats to Validity . . . . .	121
5.7	Chapter Summary . . . . .	122
<b>6</b>	<b>Conclusion and Outlook</b>	<b>123</b>
6.1	Conclusion . . . . .	123
6.2	Outlook . . . . .	125

<b>Bibliography</b>	<b>127</b>
---------------------	------------



# List of Tables

3.1	Overview of the test cases used in this work . . . . .	34
3.2	Code size (#LOC or #JVM-bytecode statements) in different phases of our approach; <i>Dif</i> = difference set (in #LOC), <i>BSlice</i> = backward slice, <i>IS</i> = intersection set, <i>Cov</i> = coverage profile, Report = final report (in #LOC) . . . . .	40
3.3	Overheads of our approach compared to full instrumentation (Full instr.) and instrumenting only the code in <i>BSlice</i> ( <i>BSlice</i> instr.); in “X/Y”, X is the run-time slowdown (a factor) and Y size overhead of instrumenting (a factor); p = passing version, f = failing version, “-” = instrumentation not possible. . . . .	41
3.4	Running times of a failing test and times for various phases of our approach (times in seconds); Total / Test is the ratio of total approach time to test time . . . . .	42
4.1	Subject programs. Column “# Unit Tests” shows the number of unit tests used in the evaluation. . . . .	56
4.2	Hadoop source code versions used in the evaluation of synthetic leaks. Column “Development Revision” shows the used revisions. Column “Changed Files” shows the differences between $V_0$ and $V_1$ in terms of files, where “m” and “a” indicate the number of modified and added files, respectively. Column “#Changed Lines” states the total number of changed lines between both versions. . . . .	57

4.3	Evaluation results of synthetic memory leaks. Column “Leak ID” indicates each synthetic leak. Section (a) reports the $LC$ value for the leaky AAS(Column “ $LC$ ”) and also the difference between the $LC$ values of the first two entries of the ranked list (Column “ $LC$ diff”). Section (b) shows the result of leak isolation: Column “rank” reports the rank of leaky AAS in the ranked list and Column “#Candidates” shows the number of allocation sites with $LC > 0$ . . . . .	61
4.4	Information related to the real memory leaks. Column “#Trig. UT(#Total UT)” shows the number of unit tests which trigger the leak pattern. The number in parenthesis indicates the total number of unit tests for that project. . . . .	63
4.5	Results of leak confidence analysis and leak isolation for real cases. Section (a) shows the result of the leak isolation: rank of the leak-inducing allocation site and the size of the ranked list of suspects with $LC > 0$ . Section (b) reports as $LC$ the leak confidence score for the leak-inducing allocation site and as $LC_{max}$ the largest leak confidence value among all sites in the ranked list. . . . .	64
4.6	The contribution of each factor in leak confidence analysis of real cases. Section (a) shows the value of each factor for the leaky allocation site of each of the cases. Section (b) reports the rank of each leaky allocation site in the ranked list using each factor. . . . .	70
5.1	Overview of studied projects. Column “Since” lists the year of the first commit and column “#Committers” presents the number of committers in each projects based on Apache Committee Information. The kLOC of each project shows the number of Java code lines retrieved by OpenHub. . . . .	84
5.2	Studied projects with statistics on number of issues (explained in Section 5.3.3). Columns “#MLeak”, “#RLeak”, and “Total” show the numbers of memory and resource leak issues per application, and their totals, respectively. . . . .	88
5.3	Cohen’s kappa measurement. . . . .	89

5.4	Distribution of detection methods for memory and resource leaks. . .	95
5.5	Taxonomy of root causes. Column “#Issues” states the total number of issues per root cause. . . . .	102
5.6	Taxonomy of repair actions. Column “#Issues” states the total number of issues per repair action. . . . .	105
5.7	Recurring code transformations and examples of code before and after the transformations. . . . .	109
5.8	The evaluation of Infer static analyzer on a sample of resource leaks from our dataset. Column “Det?” reports whether Infer could detect the defect reported in the respective issue. “Code-based detection” refers to source code-based detection. “Defect” type and “Repair” type are explained in Section 5.4.4 and Section 5.4.5, respectively. . . . .	116
5.9	Comparison of common code transformations found in our study with previous work. 27Repairs refers to [97]. . . . .	118



# List of Figures

2.1	The running example for dependence analysis. It prints the factorial 10.	11
2.2	The CFG of the running example. . . . .	12
2.3	The PDG of the running example. The solid lines and dashed lines show the control-dependence and the data-dependence, respectively. .	13
2.4	Heap space memory in Java 7. . . . .	20
2.5	Garbage collection process. . . . .	21
3.1	The workflow. . . . .	30
3.2	Stack trace of the bug reported in Issue HDFS-3856. . . . .	33
3.3	Excerpt of code changes for issue Hadoop-8689 (simplified). . . . .	36
4.1	Overview of our approach. . . . .	47
4.2	Matching algorithm for drift computation of two software versions. . .	51
4.3	Pseudo code of the Artificial Allocation Site (AAS) used as a leak-triggering defect. . . . .	58
4.4	The leak-inducing changes in the Snappy-Java. . . . .	68
4.5	Runtime and RSS overhead of subject programs. Overhead of 1 ( $y$ -axis) means that the instrumented version has twice the runtime or RSS of the non-instrumented version. . . . .	72
4.6	Distribution of leak confidence score $LC$ for the allocation sites reported in the ranked list for real leaks. . . . .	73
5.1	An issue report from JIRA. . . . .	82
5.2	Overview of the empirical study design. . . . .	83
5.3	Frequency of the leak types per project. . . . .	91

## *List of Figures*

5.4	Frequency of the detection types per leak type. . . . .	94
5.5	Heatmap of the number of modified Java source code files per project.	98
5.6	Heatmap of defect types and leak types. . . . .	103
5.7	Heatmap of relationship between root causes and repair actions. . . .	108
5.8	Heatmap of recurring code transformations and single repair actions.	110
5.9	Distribution of code churn per repair action. . . . .	112
5.10	Distribution of number of added and removed lines over the studied projects. . . . .	113
5.11	Distribution of change complexity over the repair patches. . . . .	114
5.12	Distribution of time to resolve per repair action. . . . .	115
5.13	TTR comparison of leak-related and other bugs in studied projects. .	117

# Chapter 1

## Introduction

Today’s software systems become more extensive and more complicated because of growth in size and functionality. Lehman states in his laws of software evolution [67] that the software complexity increases as it evolves. Such increasing complexity confronts software maintenance (i.e., debugging, testing, and verification) with more challenges. Fixing a bug or adding a new feature can introduce new bugs [135]. Debugging becomes harder and more expensive in cases of non-functional defects, i.e., those defects which do not violate the semantics of the functions such as memory leaks or performance bugs. These bugs often skip the functional testing phase and only become visible in the production environment. Hence, they can impose a significant economic impact. For example, a memory leak in Amazon’s EC2 cloud service caused a partial outage in October 2012, affecting operations of hundreds of EC2 customers<sup>1</sup>. In a typical software organization, the cost of debugging, testing, and verification can easily range from 50% to 75% of the total development cost [31, 47].

How can we help developers to localize the software defects more effectively during software development? The statement of this dissertation is to answer this question analytically and empirically in three aspects: 1) Scalable isolation of failure-inducing changes; 2) Automated memory leak diagnosis via version comparison; 3) Empirical study on resource and memory leaks. In the following, we describe these aspects in more detail.

---

<sup>1</sup><https://aws.amazon.com/de/message/680342>

## 1.1 Scalable Isolation of Failure-Inducing Changes

Automated debugging techniques attempt to find the causes of a program failure without or with minimal human interactions. Many previous works proposed different approaches to solving this problem [125]. Most of them report a ranking list of the suspicious code locations to the programmer. By examining them, the programmer is likely to identify the root cause of the defect.

Despite indisputable advances [11, 27, 35, 58, 69, 102], automated debugging is still facing significant challenges preventing its widespread adoption in practice [98]. Two reasons contribute to this situation. First, only pointing to the suspicious statements does not help developers to understand the root cause of the defects. In most of the cases, the root causes of the failure are in different code locations than the failure site (even in different files). Lack of precision is the other weakness of the automated debugging techniques. Even reporting only 1% of the code locations as the search space for the root causes of the defects is not precise enough in case of large applications. Such a level of specificity means that on a project with 100k lines of code (LOC), a developer still needs to examine 1000 lines to find the cause of the defect. That is beyond the average acceptance level of programmers. Most users only inspect the first page of the debugging reports [98], reducing the efficiency of such techniques for large-scale projects.

In this dissertation, we attempt to approach the latter problem. Our observation of the current software development reveals that modern software evolves through a series of minor changes resulting in many intermediate versions between the two major releases of software. Developers test each minor version thoroughly using the application’s test suite and perform code review processes before releasing a new version. This is a typical approach in modern software development, for example in the setup of continuous integration and deployment [141].

We revisit the *version comparison* technique to localize newly introduced defects in the latest development version of the software. Version comparison contrasts the behavior of two software versions under the execution of an existing test suite. We leverage static and dynamic analysis and compare the sets of statements executed in the latest (faulty) version against those executed in a previous (correct) version.



Here, we assume that at least one test case fails using the new (faulty) version, while the same test case passes using the previous version.

## **1.2 Automated Memory Leak Diagnosis via Version Comparison**

In memory-managed languages such as Java, C#, or Python, a component called garbage collector (GC) is responsible for memory management. Garbage collector allocates memory, finds the unused objects, and frees the heap memory. To find the unused objects, GC approximates the object liveness by its reachability. It means that the objects which are unused but still reachable from the root objects can skip this process. In such cases, memory leaks might occur. A common case is forgotten references from the collections [131]. For example, objects encapsulating requests to a web server are often referenced from a collection (e.g., list or map) which implements a processing queue. If the reference is not removed from the queue after the request is processed, the garbage collector cannot dispose of the associated object resulting in a memory leak.

Leaks are notoriously hard to detect, reproduce, and fix. First, there is a long latency between triggering a leak and the manifestation of visible symptoms such as memory bloat or performance degradation [49]. Second, the leaks are sensitivity to inputs and execution environments [16]. Therefore, many of such defects escape in-house quality assurance measures including unit, integration, and even performance testing. If these bugs occur in a production environment, they can impose a significant economic impact.

Leak diagnosis is an essential problem for both researchers and practitioners. An empirical study [75] on the publications of the top-tier Software Engineering conferences reveals that the detection and root cause analysis of memory leaks is among the top 10 highly rated research ideas. Several tools [38, 49, 86] and research techniques are developed and designed to help developers to detect and isolate resource and memory leaks. Most of these approaches follow a “symptom to root cause” method [130]. One strategy is to apply staleness (time since the last use of

## 1. Introduction

an object) analysis to identify “dead” objects - those objects which can not be accessed for a long time [16, 50, 60, 93, 132]. Another group of work is based on analyzing heap growth [23, 59, 111, 112]. The other direction is analysis of the captured state [28, 86, 88, 129]. Most of these works assume that a leak has been already observed and the test code triggering the leak is also available. These approaches help developers with isolating the root causes of the leaks at the cost of a proprietary execution environment (e.g., a modified JVM [16]) or significant execution slowdown (e.g., 300-400% slowdown [132]). Recent approaches for C/C++ focus on performance efficiency and promise slowdown of  $\leq 3\%$  [60].

However, none of these works address that new software projects: 1) evolve as a series of relatively small code changes, and 2) often provide an extensive test suite. In this work we exploit (1) for an anomaly detection-based approach for leak diagnosis, and (2) for triggering memory leaks during in-house testing. Our approach supports the automated diagnosis of memory leaks during software development and helps to isolate the root cause if a leak is already detected. It requires only minimal modification on the software testing framework with no changes in the source code of tests and software. This property makes our approach applicable to the existing projects. Since our method is an anomaly detection technique, it is unnecessary to execute the tests until significant memory bloat occurs (such memory bloat is a prerequisite for most existing techniques for leak detection). Therefore, the diagnosis time remains proportional to the time for executing the project’s test code.

Inspired by the delta debugging technique [138] for the isolation of “crashing” errors, we use software version comparison to uncover memory-related anomalies of the current (latest) software version. In this way, we can extract additional information which is not available when investigating each software version by itself. We use the heap growth as a symptom for leak detection. Assume two versions of a software codebase (one previous non-leaky and the current leaky version) and the existing test suite of the application. For each software version, we collect the memory usage of the code statements exercised during the execution of the application’s test suite. Afterward, we compare the memory profiles of the two

versions. Finally, we assign a suspiciousness score to each exercised code statement. The higher the score, the more likely the code statement be the root cause of a leak.

## 1.3 Empirical Study on Resource and Memory Leaks

Many academic studies, language features, and tool supports propose techniques for detecting and localizing of leak-inducing defects. However, the impact of these efforts depends on whether they target common or rare issue types, whether they can handle complicated cases, and whether their assumptions are realistic enough to apply in practice. Programming language enhancement (e.g., `try-with-resources`) or tools (e.g., FindBugs, Infer) help us only to find the resource leaks. Many of the academic works are motivated by anecdotal evidence or by empirical data collected only from small sets of defects. For example, Xu and Rountev [131] propose a method for detecting memory leaks caused by obsolete references from within object containers but provide only limited evidence that this is a frequent cause of leak-related bugs in real-world applications. As another example, Leakbot [88] introduces multiple sophisticated object filtering methods based on observations derived from only five large Java commercial applications.

A systematic empirical study of a large sample of leak-related defects from real-world applications can help both researchers and practitioners to have a better understanding of the current challenges on leak diagnosis. We believe such a study can be beneficial in the following directions: 1) A representative study can characterize the current approaches for leak diagnosis used in practice; 2) It helps programmers to avoid mistakes made by the other programmers and shows some of the best practices for leak diagnosis; 3) It can act as a verification for the assumptions used in previous work.

To the best of our knowledge, the research body of empirical studies on resource and memory leak-related defects is relatively thin in comparison with the vast body of studies about other bug types (e.g., semantic or performance bugs). The existing studies [78, 115] provide only a little information about the characteristics of detection types, root causes, and repair actions of leak defects. To fill this gap, we conduct a

detailed empirical study on 491 real-world memory, and resource leak defects gathered from 15 large, open-source Java applications.

We manually study the collected issues and their properties: leak types, detection types, common root causes, repair actions, and complexity of fix patches. Based on our findings, we draw several implications on how to improve avoidance, detection, localization, and repair of leak defects.

## 1.4 Contributions

This dissertation makes the following contributions:

- We propose and implement an approach for isolation of failure-inducing changes with leveraging static and dynamic analysis (Chapter 3). Our approach requires a failing test, a previous correct and current buggy version of the software, and a failing test. Given these requirements, first, we collect all code locations which affect the failure using backward slicing. Then, we intersect this subset of code with the code changes between two software versions to show which code locations should be instrumented. After instrumenting the intersection, we execute the failing test with both correct and buggy versions to get the coverage profile for each software version. Finally, we compare the coverage profiles to identify the statements which are likely to be the root cause of the defect.
- We propose an approach for memory leak detection based on version comparison (Chapter 4). Similar to our approach in Chapter 3, we use the changes between two consecutive versions to isolate the root causes of memory leaks. In contrast to the approach for crashing bugs (Chapter 3), we use all existing tests from the applications' test suite for leak detection. We propose a confidence measure which assigns a suspiciousness score to the code locations which allocate memory. The higher the confidence score, the more likely that the code location is a potential root cause of a memory leak. Contrary to previous approaches on memory leak detection, our approach can be used

during the development phase and before visible memory bloat occurs, in a time proportional to the execution of a test.

- We conduct an empirical study on 491 leak-related bugs from 15 mature, large Java applications to understand the characteristics of memory and resource leaks reported in bug trackers. To the best of our knowledge, this is the first work which studies the leak-related bugs from real-world applications comprehensively while using a large set of issues from diverse open-source applications. We propose taxonomies for the leak types, for the detection methods, for the root causes, and the repair actions (Chapter 5). We draw a set of implications which can help both practitioners and researchers to improve existing techniques for leak avoidance, detection, and diagnosis.

## 1.5 Papers Appeared

The content of this dissertation has appeared partially in the following papers published (or submitted) in peer-reviewed conferences and journals:

- M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak. Memory and Resource Leak Defects and their Repairs in Java Projects. Accepted to be published in *Springer Journal of Empirical Software Engineering*, DOI: 10.1007/s10664-019-09731-8 (preprint: <http://arxiv.org/abs/1810.00101>).
- M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak. Memory and Resource Leak Defects in Java Projects: An Empirical Study. In the Companion Proceedings of *ACM/IEEE International Conference on Software Engineering (ICSE)* 2018.
- M. Ghanavati and A. Andrzejak. Automated Memory Leak Diagnosis by Regression Testing. In the *Proceedings of Source Code Analysis and Manipulation (SCAM)* 2015.
- M. Ghanavati, A. Andrzejak, and Z. Dong. Scalable Isolation of Failure-Inducing Changes via Version Comparison. In the *Proceedings of*

## 1. Introduction

*International Workshop on Program Debugging at IEEE ISSRE 2013* (Best Paper Award).

The following papers are also in the line of this dissertation (not included in the dissertation):

- A. Andrzejak, F. Eichler, and M. Ghanavati. Detection of Memory Leaks in C/C++ Code via Machine Learning. In the *Proceedings of Workshop on Software Aging and Rejuvenation (WoSAR) at ISSRE 2017*.
- Z. Dong, M. Ghanavati, and A. Andrzejak. Automated Diagnosis of Software Misconfigurations Based on Static Analysis. In the *Proceedings of International Workshop on Program Debugging (IWPD) at IEEE ISSRE 2013*.

## 1.6 Overview and Organization

This dissertation is principally positioned in the domain of software testing, debugging, and evolution.

In Chapter 2, we introduce terminologies, background, and related work. In Section 2.1, we explain program analysis and survey the fault localization techniques for crashing bugs. Section 2.2 describes resource and memory management in Java, resource and memory leaks and the detection techniques.

In Chapter 3, we introduce a new technique to isolate the failure-inducing changes for functional bugs. Given two versions of a codebase, a test (which passes with the previous version and fails with the newer version), and a stack trace of the failure, we leverage both static and dynamic analyses to localize the faulty code. We evaluate our technique on four real cases. In three cases, our technique can precisely pinpoint the faulty change or at least the method containing the faulty change.

In Chapter 4, we present a new technique to diagnose memory leaks in the presence of software regression. Given two versions of a codebase and a test suite (provided with the codebase), we instrument the codebase of the program to collect the memory-related information during the execution of the tests. Then, we assign a suspiciousness score to the code locations in the collected profiles. Top-ranked entries in the ranking list are highly likely to be the potential memory leaks. We evaluate our approach on

seven real-world leak-inducing defects. Results show that our approach has sufficient detection accuracy and is useful in isolating the leaky code statements.

In Chapter 5, we conduct an empirical study on real memory and resource leak defects and their repairs collected from 491 real resource and memory leaks from 15 mature Java projects. We propose taxonomies for the leak types, for the defects causing them, and for the repair actions. We use the results of our findings to draw implications about leak detection, fault localization, and root-cause analysis.

We conclude this dissertation in Chapter 6 with a summary of the contributions and discuss possible future work.

# Chapter 2

## Background and Related Work

Automated debugging and fault localization has attracted a great deal of interest among researchers and practitioners. There is a large body of related work on debugging and fault localization. Wong et al. [125] surveyed more than 400 papers on fault localization techniques. This chapter introduces the required terminology, background, and preliminaries for this dissertation. We also survey the existing techniques for debugging and fault localization.

In the first section, we present the terms and techniques related to fault localization of functional bugs. In the second section, we describe the terminologies and techniques for resource and memory leak detection. We also describe how the mismanagement of the memory and other finite system resources might cause resource and memory leaks.

### 2.1 Fault Localization of Functional Bugs

This section provides an overview of the main techniques on debugging and fault localization of crashing bugs. First, we introduce static and dynamic program analyses. Then, we overview the related work on debugging techniques based on program analyses. Note that we explain program slicing in detail with a running example as it is the base for our proposed technique for fault localization of failure-inducing changes presented in Chapter 3.



```

1  public class Factorial {
2      public static void main (String[ ] args){
3          int num = 10;
4          long factorial = 1;
5          int count = 1;
6          while(count <= num)
7          {
8              factorial = factorial * count;
9              count++;
10         }
11         System.out.println( factorial );
12     }
13 }

```

Figure 2.1: The running example for dependence analysis. It prints the factorial 10.

### 2.1.1 Program Analysis

Program analysis is used to reason about the properties of the codebase of an application. Based on the requirement of executing an application, the program analysis can be static or dynamic. In the following, we explain each type in more detail.

#### 2.1.1.1 Static Program Analysis

Static program analysis reasons about the relationship between different code statements of a codebase without running the application. It normally is done by performing a *dependence* analysis. A graph representation named *control flow graph* (CFG) is used to make the analysis much easier and faster. In the following, we explain the dependence analysis. To illustrate the dependence analysis and related terminologies, we use a sample program shown in Figure 2.1 as a running example. This short program computes the factorial of a given integer (here 10) and prints the final result.

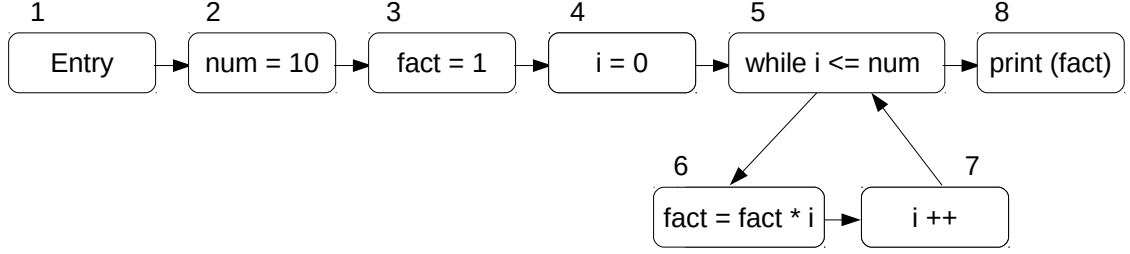


Figure 2.2: The CFG of the running example.

### 2.1.1.2 Program Dependence Analysis

Control flow graph (CFG) is a directed graph which shows the control-flow between program statements. In a control flow graph, each node represents a code statement and vertices show the control path between the nodes in the graph. Given two nodes,  $n_1$  and  $n_2$ , there is a vertex connecting  $n_1$  to  $n_2$  if  $n_2$  can immediately be executed after  $n_1$ . Figure 2.2 shows the CFG of the running example.

Vertices in a CFG show two dependency types between the nodes in the CFG. Every two connected nodes can be data-dependence or control-dependence. Given two code statements  $s_1$  and  $s_2$ ,  $s_2$  is data-depends on  $s_1$  if the data from  $s_1$  can be potentially propagated to  $s_2$ . As for control-dependence,  $s_2$  is control-depends on  $s_1$  if the execution of  $s_2$  is conditionally based on the execution of the  $s_1$ . The summary representation of the dependence analysis of a program is *program dependence graph* (PDG). Nodes in PDG are statements, predicates (such as conditions), or the special “entry” node. The entry node represents the entrance into a procedure. Figure 2.3 shows the PDG of the running example. The solid arrows show the control-dependence and the dashed lines present the data-dependence. For example, nodes 6 and 7 are control-depends on node 5. Analogously, node 6 is data-depends on nodes 3, 4, and 7 as theses nodes affect the value of node 6.

One limitation of PDG is that the PDG can only consider the dependence within the procedure calls and cannot pass the boundaries of each procedure. To determine the dependence analysis in a multi-procedural program, Horwitz et al. [52] introduced *system dependence graph* (SDG). System dependence graph is the extended version

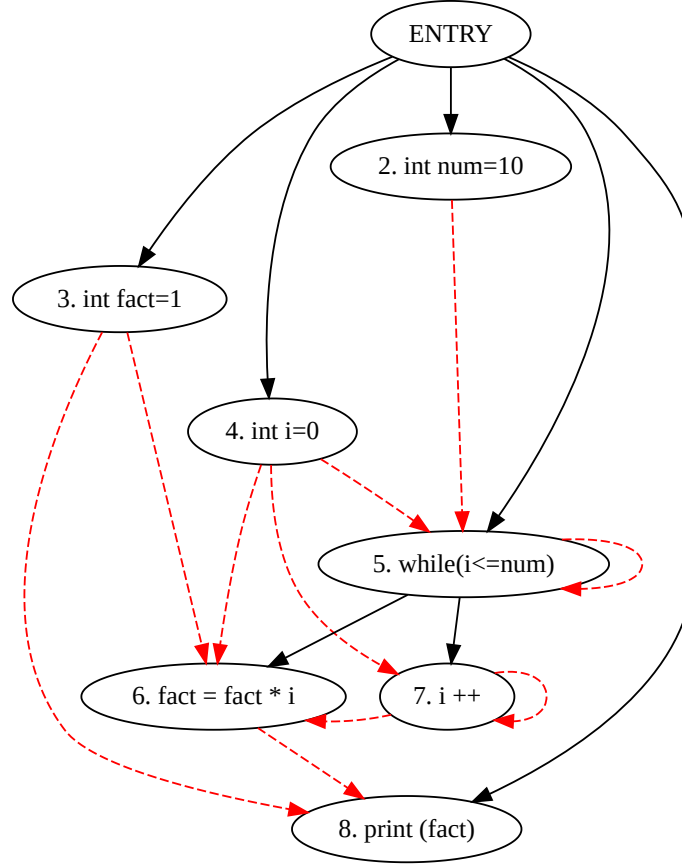


Figure 2.3: The PDG of the running example. The solid lines and dashed lines show the control-dependence and the data-dependence, respectively.

of PDG where the graph consists of the primary procedure and a set of subsidiary procedures.

### 2.1.1.3 Dynamic Program Analysis

Although static analysis techniques can considerably help programmers to debug the code, they often lack precision. Researchers mitigate this problem by leveraging the dynamic program analysis. Dynamic analysis analyzes only a single run, unlike the static analysis which practices all possible executions. Although the dynamic analysis is expensive due to overhead, it includes fruitful information about the behavior of the program during runtime. In some types of bugs such as concurrent bugs or memory-related bugs, the runtime information is required to diagnose the root cause of the defects. Among different dynamic analysis techniques, *profiling* is the most common approach. Profiling is a dynamic analysis technique in which the

## 2. Background and Related Work

user monitors and collects the execution information during runtime. The tool which performs the profiling is called *profiler* and collect different information such as memory usage, the frequency of function calls, and others. Using profiling, programmers can achieve precise runtime information. One of the main techniques for profiling is *code instrumentation*. In this approach, some new code (instructions) will be added to the source code or the program binary. Instrumentation allows programmers to trace the code statements which are executed during runtime. Instrumentation can be performed offline and online. In offline instrumentation, instructions will be added to the bytecode of the program or a compiled executable. In online instrumentation, the instructions will be added to the code directly before code execution or the code will be modified during execution.

### 2.1.2 Fault Localization Techniques

In the following, we present the main techniques for fault localization. The main directions are slicing-based approaches, program state-based approaches, spectrum-based techniques, and statistical debugging.

#### 2.1.2.1 Program Slicing-Based Approaches

Dependences can help us to focus on a specific part of the program being debugged. Weiser in his pioneering work [124] leveraged the dependence graph to introduce *program slicing* for debugging task. The core idea of the program slicing in the debugging is to find a subset of code statements which contribute to the buggy statement (i.e., the point of interest). The subset is called *slice* and the point of interest is called *slicing criterion* or *seed* statement. The slicing criterion is often the code statement in which the program has crashed and can be obtained, for example from the stack trace at the failure site. With leveraging a program dependence graph and the seed statement, slicing can identify a set of statements which affect the value of seed statement. This slicing set called a *backward slice*.

A backward slice is defined as follows: For a given program  $P$  and a statement  $s$  with a variable  $v$  at the program location  $l$ , a *backward slice* computed from  $s$  contains all of the statements in  $P$  which can affect the value of  $v$  [124]. It is obvious that if

$l$  is the failure site, a backward slice includes all of the statements which might be the root cause of the failure at the code location  $l$  or at least contains information about the failure. Take our running example and consider the print call at line 11 as the slicing criterion (seed). In this case, all lines of the program are present in the backward slice.

### 2.1.2.2 Extension of Program Slicing-Based Approaches

A traditional slicing often produces a too large slice, especially in real large applications. However, not all of the statements in the slice are equally relevant to the slice criterion from a programmer’s perspective. Different approaches extended the slicing technique to reduce the size of the slice further to overcome the size of static slices. Dicing, chopping, and thin slicing are some of these techniques.

Dicing [77] is a set difference of two static slices, one for the incorrect variable and one for the correct variable. Chopping [56] is an intraprocedural slicing which combines the intersections and unions of the forward and backward slices.

Sridharan et al. proposed *thin slicing* [113] to overcome the large size of the static slices. Thin slicing is a slicing technique based on value-flow relevance. The idea of thin slicing is to exclude the statements which contain less information about the value of the seed statement.

A thin slice only includes those statements from the codebase which may “copy-propagate” the value to the slicing criterion [113]. Based on this definition, the statements in the traditional slice are divided into two types: the producer and the explainer. Statement  $s$  is a producer statement if it flows a top-level value to the seed statement. Explainer statements reason about the effects of a producer statement on the seed statement. A thin slice only contains the producer statements.

Contrary to the traditional full slicing, the thin slice is not executable. However, it can be used in combination with other tools and techniques in tasks such as debugging. The size of the slice is considerably smaller than the traditional slice, and it identifies the most relevant statements to the seed statement more effectively.

Static slicing contains all statements affecting the value of the seed statement. It can also include statements with no correlation to the failure site. Dynamic slicing is introduced to reduce the slice size by incorporating the runtime information [2,

63]. Many approaches used dynamic slicing as a standalone, improved version, or in combination with the other techniques for the purpose of fault localization [1, 4, 33, 73, 114, 120, 126, 142–146].

### 2.1.2.3 Program State-Based Approaches

In modern software development, developers add, remove, or modify the code elements to answer the need of the end-users or to improve the software quality. In large projects, developers push thousands of commits to the codebase repositories per year. For example, Eclipse IDE for JAVA shows a 12-month commit rate of 15546 in the time of writing this dissertation<sup>1</sup>. Although the goal of these changes is to improve software quality, they can break some code functionalities as well. The software may not work anymore after applying the changes even with similar input, and environmental setup. Therefore, fault localization in such cases can become a time-consuming and tedious effort.

Different approaches are introduced to localize the bugs induced by software changes. Zeller et al. proposed *delta debugging* in work “Yesterday, my program worked. Today, it does not. Why?” [139]. Assuming the presence of some changes and a bug induced from these changes, delta debugging tries to find the failure-inducing change. In principle, it uses the idea of *divide-and-conquer* algorithm to find the faulty change by gradually narrowing down the search space. Zeller later applied delta debugging to multiple program states to isolate the variables and values that caused the failure [140]. Other approaches explored the potential of delta debugging to improve the accuracy of fault localization further. Gupta et al. [46] combined program slicing with delta debugging. Despite the power of delta debugging in fault localization, it suffers several drawbacks, mainly high false positives and the need for many test cases — these problems addressed by other works [87, 137].

### 2.1.2.4 Spectrum-Based Approaches

Spectrum-based fault localization (SBFL) approaches are widely used for fault localization. They collect the execution information of failing and passing test cases

---

<sup>1</sup><https://www.openhub.net/p/eclipse/commits/summary>

and compare them to find the root cause of the failure. Then, they assign a suspiciousness score to the code elements using a formula. The higher the score, the more likely the code element is to be faulty. Many approaches tried to introduce a new SBFL formula. More than thirty formulae exist as shown by Wong [125]. Despite the huge research body on spectrum-based fault localization techniques, their effectiveness is still in question. Yoo et al. [136] show that there is no single SBFL formula which outperforms others. Pearson et al. [99] show that we cannot predict the effectiveness of SBFL formulae on real faults by applying them to artificial bugs.

#### **2.1.2.5 Statistical-Based Approaches**

Statistical debugging is introduced in pioneering work by Liblit et al. for bug isolation [71]. It analyzes a large number of executions gathered from running the instrumented application and rank the suspicious predicates which are highly relevant to the bug. Correctly evaluated predicates only executed in the failing executions are likely to be more suspicious. Liu et al. [72] introduced SOBER as another statistical debugging approach in which a predicate can be evaluated multiple times as true in a single execution.

Many approaches are developed based on the pioneering works on statistical debugging [70, 71]. Some approaches adopted different types of predicates [7, 10, 27, 44]. For instance, Holmes et al. [27] extended the statistical debugging with incorporating the path profiles as predicates for bug isolation. Statistical debugging is also used in other fields such as detection of performance problems [110].

#### **2.1.2.6 Other Approaches**

There exist many other approaches for fault localization such as model-based [32, 80, 127], machine learning-based [18, 19, 36], data mining-based [22, 89], formula-based techniques [13, 100]. In model-based techniques, a correct available version of the application is required to be used as an oracle. The current program profile will be contradicted to this oracle to find the potential bugs. Machine learning-based approaches explore computer program algorithms such as classification to provide a

## 2. Background and Related Work

model based on a sample dataset. Data mining can also help to determine a healthy model of a program from existing data of the program in question (e.g., from version control systems such as GitHub or SVN). Banerjee et al. [13] generate an alternative input (which differs from failing input in the control flow behavior) and then compare the program executions with this input to find the root cause of the failure.

## 2.2 Resource and Memory Leak Detection in Java

Mismanagement of memory or other finite system resources such as threads or I/O streams can result in a software failure. Depends on the programming language and the type of the resource, the resource and memory management is different. In unmanaged-languages such as C/C++, the programmer is responsible for resource and memory management. However, in managed languages such as Java, resource and memory management are treated differently. In Java, the management of finite system resources such as threads, I/O streams, or database connections is of the responsibilities of programmers while a specific component called garbage collector mainly performs memory management. In this section, we first describe the memory management in Java. Then, we define the terms resource and memory leaks. Finally, we sketch the related work on detection, prevention, and repair of resource and memory leaks.

### 2.2.1 Memory Management in Java

One of the tasks of memory management in Java is to find unused objects and freeing the occupied space to make it available again for allocation by other objects. In some programming languages such as C/C++, the programmer is responsible for releasing the memory after the usage. The following code snippet shows an example of memory allocation and deallocation in C/C++:

```
# Allocating a char variable of size 10
p = (char*)malloc(sizeof(char) * 10)
# Releasing the allocated memory
free(p)
```



With the `malloc` keyword, a pointer of type `char` with a size of 10 is created and allocated. The `free` keyword explicitly deallocates the used memory.

Contrary to the explicit memory management, in memory-managed languages such as Java, Python, and C#, we benefit from automated memory management. In Java, the objects are stored in a space called *heap*. The references to the objects are held in another memory space called *stack*. Following code statement shows a memory allocation for a `String` object in Java:

```
String str = new String ("Allocate Memory")
```

With the keyword `new`, we create an object of the type `String` in the heap space. This object is referenced via the “`str`” variable stored in the stack.

Each Java application owns an instance of Java Virtual Machine (JVM). At the startup of JVM, the heap memory will be created. The heap memory consists of two main parts: the young and old generations. Figure 2.4 shows the different parts of the heap. The *Eden* (or young generation) is the space where the memory is initially allocated. If these objects remain in the JVM for a while, they will be moved to another part called *Tenured generation* (or old generation). There is another memory part in JVM called *permanent generation*<sup>2</sup> which stores the application metadata used by JVM to describe the classes, methods as well as threads. The heap memory is shared among all threads running on the JVM process. During program execution, the heap size varies based on object allocation and deallocation. The size of the heap is configured and set before starting the JVM. If the process requires more memory than the maximum heap size defined at configuration time, the JVM throws an out-of-memory error.

To mitigate the out-of-memory error, a program called *Garbage Collector* (GC) is responsible for automated memory management in Java. GC allocates memory, finds the unused objects, and frees the heap memory for making space for the creation of new objects. Figure 2.5 visualizes the process of the garbage collection in Java. Each time a new object is initialized, garbage collector allocates the memory for that object in the Eden space of the heap (Figure 2.5 (a)). The size of the objects depends on the object initialization. The objects in the heap space can be referenced by other objects

---

<sup>2</sup>The permanent generation is deprecated after Java 7 and is replaced by another component called *Metaspace*.

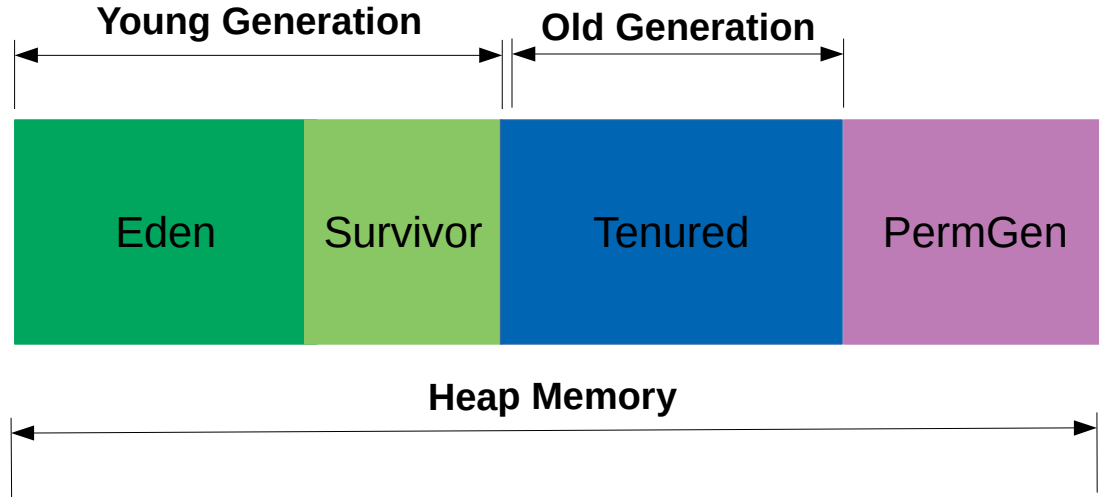


Figure 2.4: Heap space memory in Java 7.

inside and outside the heap. Objects outside the heap are called root objects and can be running threads, local variables, JNI<sup>3</sup> variables and others. When the Eden space becomes full (Figure 2.5 (b)), GC performs a *minor* garbage collection. If an object survives the garbage collection, GC moves them from Eden to the survivor space  $S_0$ . After second minor garbage collection, GC moves existing objects from Eden to the  $S_1$  as well as those objects in  $S_0$  to  $S_1$ . If an object survives several iterations of garbage collection, GC assumes the object as a long-living object and moves it to the old generation (tenured). When the heap becomes full, GC performs a so-called *major* collection.

In the major collection, the garbage collector tries to find the objects which are not used by the application (Figure 2.5 (c)). For this purpose, GC should reason about the liveness of the object. An object is *alive* if it is still used by the JVM process, otherwise, it is a *dead* (unused) object. Identifying the liveness of an object is not easily feasible. Therefore, the GC approximates the object liveness via its *reachability* from the root objects. An object is reachable if there is any path to the root objects. In other words, an object will not be reclaimed if it is still reachable by a chain of references from one of its roots. The roots of an object can be current call stack(s),

---

<sup>3</sup>Java Native Interface

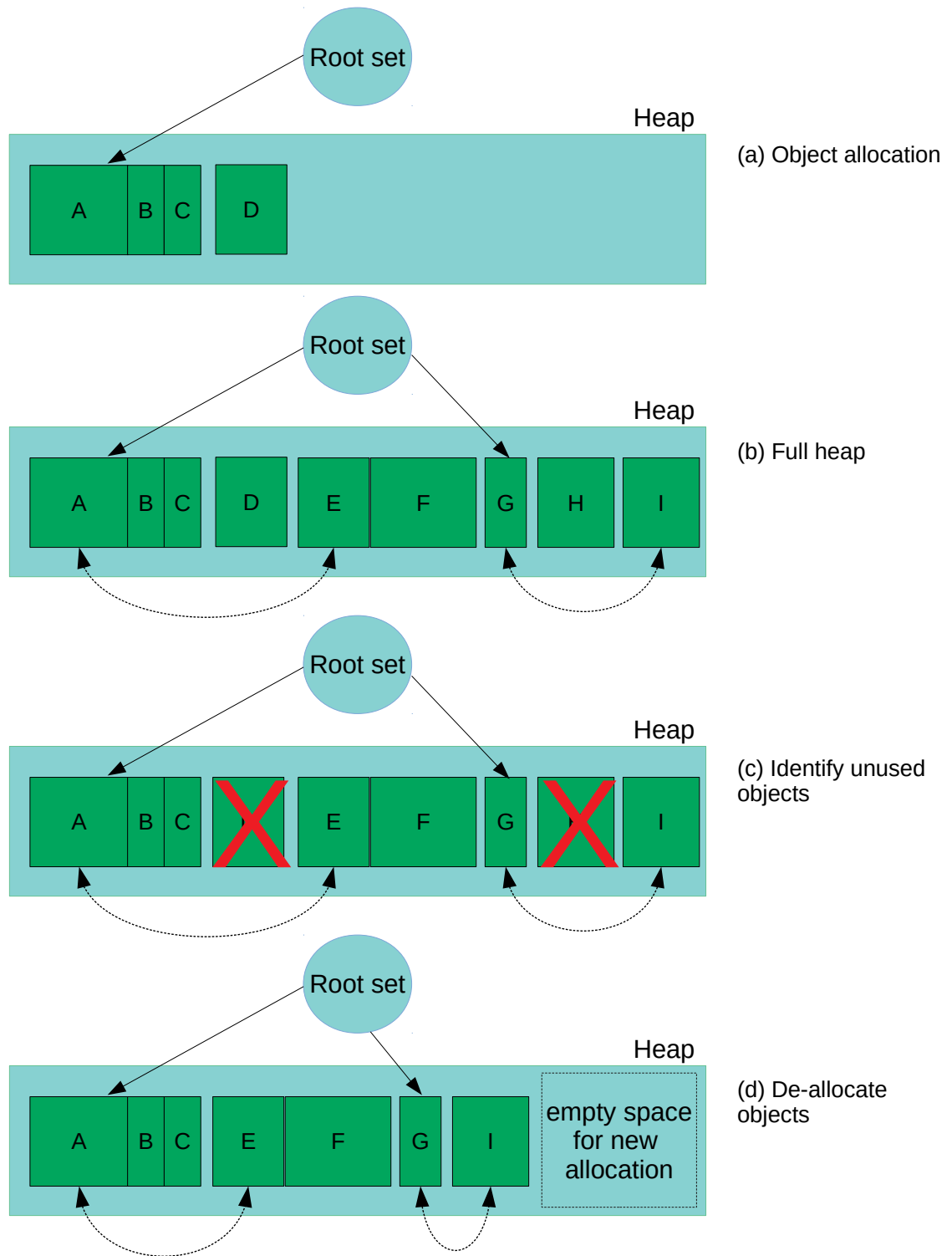


Figure 2.5: Garbage collection process.

## 2. Background and Related Work

references in registers, global variables, classloaders, live threads. The unreachable objects are then garbage collected by the GC (Figure 2.5 (d)).

### 2.2.2 Resource and Memory Leaks

Leaks occur due to mismanagement of memory or finite systems resources. In the following, we explain these two types.

**Memory leaks.** Contrary to the unmanaged languages such as C or C++ in which programmer is responsible for freeing the memory, in memory-managed languages such as Java or C#, the garbage collector reclaims the space. A programmer can rely on the garbage collector to release references due to dangling pointers (references to already freed objects) or lost pointers (lost references to unreleased objects). However, if the references to the unused objects are present in the running process, they cannot be garbage-collected. As a sequence, a memory leak might be triggered. In other words, a *memory leak* in Java occurs when a process maintains unnecessary references to some unused objects.

**Resource leaks.** In Java, finite system resources like connections, threads, or file handles are wrapped in special *handle objects*. Programmer accesses such a resource by normal object allocation. However, in contrast to memory management, the developer should dispose of a system resource by making an explicit call to the disposal method of the handle object (or by ensuring that a thread has stopped). Besides this, all unnecessary references to such objects should be removed to prevent the potential memory leak. Hence, a *resource leak* occurs when the programmer forgets to call the corresponding close method for a finished handle object. Similar to the memory leak, resource leaks gradually deplete system resources which degrade performance and can lead to failure.

### 2.2.3 Debugging Leak-Inducing Defects

The problem of memory leak attracted a great deal of interest in the last years. The body of research is broad, from leak avoidance and detection to the repairing of leak-inducing defects. In this section, we discuss the main research directions in this field.

### 2.2.3.1 Software Rejuvenation

Software aging is progressive performance degradation due to resource depletion which causes severe impacts on software quality. Controlling the effects of software aging defects is one of the main approaches to reduce the impact of software aging. Software rejuvenation [53] is a proactive approach to avoid software degradation via scheduled restarts. Research here include case studies [8, 68, 108], modeling of performance degradation [6, 21, 40, 117], and limiting the application downtime due to the scheduled restarts [3, 20, 107].

Matias et al. [81] proposed a systematic approach to detecting software aging in shorter test time and higher accuracy compared to traditional aging detection techniques via stress testing and trend detection. The approach is based on a comparative differential analysis where a software version under test is compared against a previous robust version by observing the behavioral changes during system tests of different resource metrics.

### 2.2.3.2 Leak Detection via Static Analysis

Static analysis has been used predominantly for languages with manual memory management like C/C++. It can detect defects such as double or missing calls of `free()`. Techniques include e.g., reachability analysis via a guarded value flow graph [25], backward data-flow analysis [95], or detecting violations of constraints on object ownership [51]. A major problem of static analysis is the lack of scalable and precise reachability/liveness analysis for heap objects in managed languages like Java. A recent approach named LeakChecker [133] attempts to overcome this problem by focusing on loops specified by the developer.

Many approaches have proposed to detect resource leaks in Java and C [26, 34, 106, 116, 121]. They usually use static analysis techniques to find the unclosed resources in different execution paths. There is also research on resource leak detection in Android [12, 45]. The main goal in these approaches is to find any execution paths from the opened resource in which the used resource is not closed.

Using code transformation and static approximation of resource lifetime, CLOSER [34] determines the higher-level resources which contain references to other resources

in the source code of the application. Then it inserts disposal calls at appropriate points in the source code of the application to release the resources with expired lifetime.

### 2.2.3.3 Leak Detection via Dynamic Analysis

The static-based approaches often suffer from a lack of precision and introduce many false positives. To mitigate this problem, researchers leveraged dynamic analysis for memory leak detection, especially in memory-managed languages. The major directions of dynamic leak detection are: staleness detection [14, 50, 61, 94], growth analysis [37, 41, 59, 81, 111], analysis of captured state [29, 88, 129], and hybrid approaches [101, 131].

Staleness (i.e., lack of recent read/write accesses) is the most accurate property of leaked memory. It has been used firstly in a pioneering work by Hauswirth and Chilimbli [50]. The key problem in dynamic analysis-based approaches is the overhead of monitoring object accesses. Several approaches have been proposed: path-biased sampling [50], page-level sampling [93], modifications of the JVM [16], or focusing on a specific data structure [132]. A recent work Sniper [60] can reduce the total runtime overhead for C/C++ to less than 3% by exploiting hardware units in modern CPUs. For Java, the lowest overhead is still about 80% [132], despite that only containers are monitored, and code annotation is used.

Memory leak also can be detected by looking at the memory usage. The growth of memory usage during the application runtime can point to potential memory leakage. Cork [59] finds the growth of heap data structures via a directed graph where each object traces all references pointing to itself. FindLeaks [23] tracks object creation and destruction and if more objects are created than destroyed per class (i.e., number of residuals grow), a suspect is found (runtime overheads are not reported). Previous work [111, 112] and some modified versions of the NetBeans Profiler<sup>4</sup> exploit the observation that for a perpetually leaking class many different generations of its instances exist. Machine learning techniques help here for low latency leak detection (with runtime overhead of about 40% [112]).

---

<sup>4</sup><https://profiler.netbeans.org/>

LeakBot [88] uses complex, multi-phase object ranking to find suspicious regions of heap objects which grow over time (using multiple heap snapshots). LeakChaser [129] exploits invariants among lifetimes of objects. It requires a developer to determine regions or objects belonging to the same “transaction” in order to detect invariant violations. LEAKPOINT [28] uses dynamic tainting to track heap memory pointers. It is implemented on top of Valgrind [90] which results in a runtime overhead of 100–300 times. The Valgrind tool Omega [86] uses related approaches with similar overheads. Xu and Rountev [131] target memory leaks caused by collections and try to rank the suspicious code locations by assigning a leak confidence value based on staleness and memory usage. Perfblower [37], a new domain-specific language (ISL) is introduced to describe the memory-related performance problems which can be observed during a heap history update. A new compiler will compile the generated ISL. Then with running the instrumented version of the code, the target class will be amplified during runtime when the symptom of the memory leak is observed.

Some work introduced approaches to tolerate the memory leaks by keeping the program in a running state [15, 17, 101]. They achieve this by reducing performance degradation (e.g., with predicting and reclaiming the leaky objects at runtime). However, this is not the final repair and developers still need to fix the underneath leak defect.

#### **2.2.3.4 Leak Detection via Version Comparison**

A few works utilized the idea of comparing different software versions to reason about the existence of memory leaks. Langner and Andrzejak [64] and Matias et al. [82] use cumulative memory consumption metrics (such as heap usage or residual set size) for memory leak detection. Langner and Andrzejak [64] suggest a simple visual detection technique. Matias et al. [82] evaluate a more sophisticated anomaly detection method, in particular, control charts.

Langner and Andrzejak [65] compare the memory usage of the integration or unit tests between two software versions to localize the memory leak. It provides a ranking list based on the type of allocation sites and the number of residual objects. In this approach, new allocation sites (i.e., those allocation sites which only exist in the newer version of the software) have a higher rank than the existing allocation sites (i.e., those

allocation sites exist in both older and newer versions). For each type of allocation sites, the higher ranks are given with comparing the number and cumulative size of residual objects per allocation site.

### 2.2.3.5 Automated Leak Repair

Recently, automated program repair (APR) attracted the attention of researchers. The goal of the APR is to suggest the patch candidates automatically which passes the tests successfully. Pioneering work GenProg [123] introduces a patch generation technique based on a genetic search algorithm. Kim et al. [62] propose an automated program repair technique based on patterns learned from real patches. It generates correct patches for 27 out of 119 bugs. All the provided fix patterns are one-line statements. Prophet [76] learns the properties of successful patches to guide finding the appropriate candidates. HDRepair [66] leverages information derived from the history of the previous patches of hundreds of Java projects to select the correct patch candidates. All the mentioned techniques differ in defining the search space and the approach to find the correct patch. Semantic-based techniques [84, 91] have also been explored. Angelix [84] extracts semantic constraints from the application codebase and generates fixes using program synthesis.

Automated leak repair (i.e., providing fix candidates for leak-inducing defects) is still new, and the body of the related work is thin [39, 74, 118, 134]. Footpatch [118] generates fixes for resource leaks in C and Java as well as fixes for memory leaks in C. However, it is not able to detect memory leaks in Java. Hybrid approaches [39, 134] leverage static and dynamic analyses to fix memory leaks in C. They analyze the execution paths of each allocation/deallocation and insert `free` when no release is encountered. In work [74], two repair patterns (*AddFree* and *MvFree*) are used to provide correct patches for 16 out of 41 memory leaks in C.



## Chapter 3

# Scalable Isolation of Failure-Inducing Changes

Despite indisputable progress, automated debugging methods still face difficulties in terms of scalability and runtime efficiency. To reach large-scale projects, we propose an approach which reports small sets of suspicious code changes. Its strength is that the size of these reports is proportional to the number of changes between code commits, and not the total project size. In our method, we combine version comparison and information on failed tests with static and dynamic analysis.

We evaluate our method on real bugs from Apache Hadoop, an open source project with over 2 million lines of code<sup>1</sup>. In 2 out of 4 cases, the set of suspects produced by our approach contains precisely the location of the defective code (and no false positives). In another case, we can pinpoint the method containing the faulty change. Moreover, the time overhead of our approach is moderate, namely three to four times the duration of a failed software test.

### 3.1 Introduction

Debugging is an expensive and time-consuming task in the software development process. According to studies, half of the programming time of the developers is dedicated to investigating and correcting bugs. The total cost of testing and

---

<sup>1</sup>On September 14, 2013, Ohloh (<http://www.ohloh.net/p/Hadoop>) was reporting that Apache Hadoop has 2,280,391 lines of code.

### 3. Scalable Isolation of Failure-Inducing Changes

debugging of the software development can easily range from 50 to 75 percent of the total development cost [141]. For these reasons, automated debugging has attracted a great deal of interest.

Techniques of automated debugging attempt to find the causes of a program failure without or with only minimal human involvement. In practical terms, after the analysis of data obtained from testing results and code instrumentation, a programmer is supplied by a ranked list of suspicious code locations. By examining these locations, she can identify the code fragment bearing the actual defect.

Despite of indisputable recent advances [11, 27, 35, 58, 69, 102], automated debugging is still facing significant challenges preventing its widespread adoption [98]. One of the essential ones is that while excelling at fault localization, they usually do a poor job in facilitating fault understanding. However, even knowing a (potential) fault location still requires the developer to find out what could happen there - a cognitively demanding task.

The second weakness of automated debugging is its limited scalability in terms of program size. Here even pinpointing the 1% of code which might contain the defect is not precise enough. Such level of specificity means that on a project with 100k lines of code (LOC), a developer still needs to examine 1000 suspicious lines to find the root cause of the defect. This is beyond the normal acceptance levels of programmers (most users do not inspect search results after the first page [98]), significantly reducing the utility of such techniques in large-scale projects.

We attempt to approach the second problem by narrowing our focus to scenarios where software is developed through a series of minor changes, with each intermediate version being tested thoroughly. This method is a typical approach in the development of current software projects, for example in a setup of continuous integration and deployment [141]. Consequently, this assumption is not a severe limitation, since our technique targets large-size projects.

#### 3.1.1 Core Idea

Our basic idea is to use *version comparison* to localize newly introduced defects in the latest development version. Version comparison contrasts the behavior of two software versions under the same unit and (or) integration tests. In more detail,

using static and dynamic analysis, we compare the sets of statements executed in the latest (faulty) version against those executed in a previous (base) version. We assume here that the faulty new version has caused a unit/integration test to fail, while the same test succeeded on the base version.

The key advantages of our approach are its precision and efficiency. Assuming that only the recently changed code contains the defect (this is not always but usually right), we can reduce the set of suspicious statements to a few lines of code (Section 3.4). The size of this set depends primarily on the *size of changes* (i.e., the number of differences between commits) and not on the total project size. Therefore, our technique is likely to scale and present the developers a small set of suspicious statements - even for large projects.

Secondly, we show in Section 3.4.3 that the overhead introduced by our approach is moderate. The necessary additional execution cost comes from the requirement to execute a (crashing) test on each of the instrumented base and the instrumented faulty version. Due to very sparse instrumentation, the execution time of the instrumented versions is almost identical to the original versions. As shown in Table 3.4, the total time of executing our approach is about 2.5 to 3.5 times the duration of the failed test.

### 3.1.2 Contribution

This chapter makes the following contributions:

- We propose and implement an approach to isolate failure-inducing changes by comparing subsequent versions of the software under development. It combines static analysis (backward slicing) and information on the changed code to indicate which code locations should be instrumented. A subsequent dynamic analysis (code coverage of a failing and passing version) reveals the statements which are likely to contain the defects (Section 3.2).
- We evaluate our approach on real defects from a large-scale software project, namely Apache Hadoop (Section 3.4). The results show that it can pinpoint the defective statements with high precision.

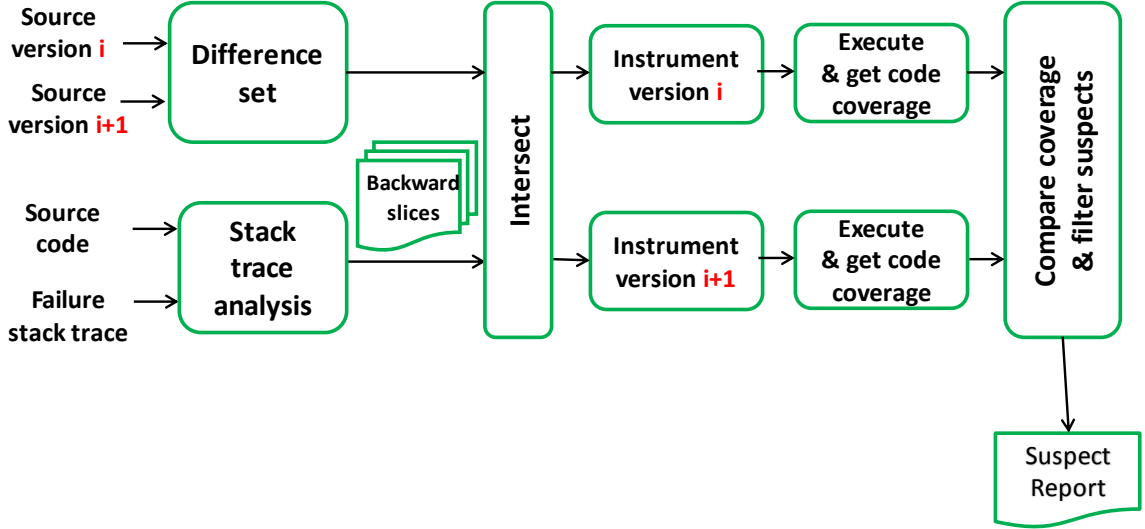


Figure 3.1: The workflow.

- We also compare the execution overhead of the images instrumented according to our approach against alternative instrumentation schemes. Our results show that the overhead of the dynamic analysis proposed by us is negligible (Section 3.4.3).

## 3.2 Version Comparison Approach

This section describes the details of our approach. Figure 3.1 shows the framework of the approach. As indicated in Section 3.1, we assume that software under development evolves as a series of versions, each one checked by executing one or more test suites. We trigger our automated debugging approach on the event that some test  $T$  has failed while executing the latest software version. We denote this latest version as  $v_f$  and call it a *failing* version. From the repository, we also retrieve a previous software version  $v_p$  (called *passing* version) which passes  $T$  successfully. Usually, this is a version directly preceding  $v_f$ .

---

**Algorithm 1:** Approach algorithm.

---

**Data:**  $v_p$  and  $v_f$ **Result:**  $SuspectSet$ **Step 1:** Find the differences of versions  $v_p$  and  $v_f$ :

$$Dif = \Delta(v_p, v_f)$$

**Step 2:** Retrieve failure site  $f_{site}$  from the stack trace**Step 3:** Compute backward slices for each version:

$$\begin{aligned} BSlice_p &= \text{BackwardSlice}(v_p, f_{site}), \\ BSlice_f &= \text{BackwardSlice}(v_f, f_{site}) \end{aligned}$$

**Step 4:** Compute the intersections  $IS_x = BSlice_x \cap Dif$  and instrument versions:

$$\begin{aligned} v_{p,inst} &= \text{Instrument}(v_p, IS_p), \\ v_{f,inst} &= \text{Instrument}(v_f, IS_f) \end{aligned}$$

**Step 5:** Execute test  $T$  on each  $v_{p,inst}$  and  $v_{f,inst}$  to get code coverage profiles:

$$\begin{aligned} cov_p &= \text{Run}(v_{p,inst}), \\ cov_f &= \text{Run}(v_{f,inst}) \end{aligned}$$

**Step 6:** Get list of suspects by applying filtering lemmas:

$$SuspectSet = \text{FilteringLemmas}(cov_p, cov_f)$$


---

### 3.2.1 Approach Overview

The steps of our approach are explained below and illustrated in Algorithm 1. First, we retrieve the set of changes between  $v_p$  and  $v_f$ , i.e.,  $Dif = \Delta(v_p, v_f)$  and call it a *difference set*. We used tools provided with software configuration management systems like SVN, Git, CVS to find the difference between two software versions.

As a consequence of the failure of  $T$  on  $v_f$ , the JVM (or operating system) provides a stack trace which is analyzed by our approach. We call the code location referenced by the top-level entry within this trace (yet not devoted to exception handling) a *failure manifestation site* or just *failure site*  $f_{site}$ . We use  $f_{site}$  as the seed statement

### 3. Scalable Isolation of Failure-Inducing Changes

to compute (for each version  $v_p, v_f$ ) the *backward slice*  $BSlice_p, BSlice_f$  [113, 124]. Essentially, it is the set of code statements which could have affected variable values at the failure site.

Subsequently, we compute (for each version  $v_p, v_f$ ) the intersection  $IS_x$  of the backward slice  $BSlice_x$  and the difference set  $Dif$  as  $IS_x = Dif \cap BSlice_x$  ( $x \in \{p, f\}$ ). This intersection gives us statements and method names which are likely to contain the defect. In the next step, we instrument the function calls within this intersection  $IS_x$  for both passing  $v_p$  and failing  $v_f$  version.

In the next step, we re-execute the test  $T$  on both instrumented versions of  $v_p$  and  $v_f$  and record coverage information. The results are *coverage profiles*  $cov_p$  and  $cov_f$ , which report those code locations that has been executed in the respective version. The last step involves comparison and filtering of the coverage profiles. The following statements are included in the set of suspects:

1. All statements *added* to or changed in  $v_f$  which are in the *failing coverage profile*  $cov_f$ .
2. All statements *deleted* from  $v_p$  which are in the *passing coverage profile*  $cov_p$ .

Finally, the resulting list of suspicious statements (suspects) together with their locations is reported to the developer as the potential root causes of a test failure.

We exemplify our approach on a real defect from the Apache Hadoop project (Section 3.4.1).

#### 3.2.2 Discussion

In the following, we discuss some secondary aspects of our approach.

As mentioned in Section 3.2.1, we examine the stack trace of the test execution on  $v_f$  to retrieve the failure site. However, we cannot take the first entry of the stack trace as this frequently points to logger code (or some exception-handling code). Therefore, we use a heuristic and check the stack trace entries (from the topmost one) if they point to the related codes to the failure, i.e., non-library and functional code. Figure 3.2 shows an example of the stack trace for issue Hadoop-3856 (Section 3.4.1), where the second topmost entry points to assumed failure site.

```
org.apache.hadoop.ipc.StandbyException: Operation category READ is not supported at
the BackupNode

at org ... $BNHAContext.checkOperation(BackupNode.java:443)

at org ... FSNamesystem.checkOperation(FSNamesystem.java:759)

at org ... system.getServerDefaults (FSNamesystem.java:1019)
...
```

Figure 3.2: Stack trace of the bug reported in Issue HDFS-3856.

The other aspect which requires explanation is slicing. For a given program  $P$  and statement  $s$  with a variable  $v$  at the program location  $\ell$ , a *backward slice* computed from  $s$  contains all of the statements in  $P$  which can affect the value of  $v$  [124]. It is obvious that if  $\ell$  is the failure site, a backward slice includes all of the statements which might be the root cause of the failure at  $\ell$ .

However, traditional full slicing [124] generates a too large slice, especially in real large-scale applications. To address this issue, we use thin slicing [113] which only contains statements that directly affect the value of the seed statement.

### 3.3 Experimental Design

We implement our approach on the top of WALA [119] which is a static analyzer developed by IBM. Slicing and instrumentation are the features of WALA which makes it useful for our approach. We implement our approach in two parts: static analysis and dynamic analysis. Finding version differences and computing backward slice is performed in the static analysis section. Slicing is implemented using WALA. For each application, we use WALA to build the corresponding call graph. Then, we compute a backward slice using this call graph and the failure manifestation point as the seed statement. For compatibility reasons, we modified some parts of WALA source code.

Dynamic profiling in our approach is implemented by Shrike<sup>2</sup> which is a third-party library for instrumenting Java byte-code provided by WALA. We use Shrike to

<sup>2</sup>[http://wala.sourceforge.net/wiki/index.php/Shrike\\_technical\\_overview](http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview)

Table 3.1: Overview of the test cases used in this work

Issue	Broken by Issue	Failing Test Case
HDFS-3856	HADOOP-8689	TestHDFSServerPorts
HDFS-4887	HDFS-4840	TestNNThroughputBenchmark
HDFS-4282	HADOOP-9103	TestEditLog.testFuzzSequences
Yarn-960	Yarn-701	TestBinaryTokens, TestMRCredentials

instruments the output statements from static analysis phase by using the predefined instrumenting schema. Due to the flexibility of instrumentation, we can easily exclude instrumenting of some parts of the code which is not necessary, e.g., Java libraries.

We execute all our experiments on a 2.9 GHz Intel Dual Core laptop with 8 GB physical memory (4 GB for the JVM), running Linux.

## 3.4 Experimental Evaluation

Our evaluation tries to answer the following research questions:

***RQ1.* How accurate is our approach to locating failure-inducing changes?**

Here we want to evaluate whether our approach can find the true defect location (sensitivity), and how many false positives are reported in the final report (specificity). We show in Section 3.4.1 the results for four test cases used in this study (Table 3.1). They demonstrate that in two of our four test cases we could find the true root cause of the failure without false positives. Analysis of another case indicates that by small extensions of our method the defect location could be narrowed to 20 LOC. We also discuss the real bug fixes of each issue as stated in the Hadoop bug log.

***RQ2.* Are there any alternatives to our approach which are simpler yet have comparable accuracy?**

This question targets the practicability of implementing our approach and tries to answer whether a more simple variant of the method could yield similar results. The brief analysis in Section 3.4.2 indicates that all of the steps shown in Algorithm 1 are necessary to achieve this level of specificity.



**RQ3. What is the time overhead of our approach, and how does it compare to alternatives?**

We have collected for each of the test cases execution times and code size in various phases of our approach (Section 3.4.3). This data is used to evaluate the overhead by two performance metrics, runtime slowdown and size overhead (Table 3.3) and to compare them against alternative approaches. We also show the total time of our method (Table 3.4).

### 3.4.1 Case Studies

In this section, we try to answer question RQ1. To this aim, we use the Apache Hadoop as a real-world, complex and large-scale project. It deploys test cases and frequent versioning which fits our requirements. We use *real* bugs from Hadoop issue tracking<sup>3</sup> system between 15th August 2012 and 27th July 2013. We select real defects according to the following criteria:

1. The failure should be caused by a Hadoop component and manifest via a Hadoop test case. In other words, we do not consider library issues or other artifacts.
2. The bug should be well documented, and it should be clear which update or patch caused the test to fail. We require this information to validate the result of our approach.
3. There should exist a passing version, i.e., a (previous) version which passed the test which failed in a subsequent (failed) version.

#### Issue HDFS-3856

The first issue is an attempted solution to a new feature request HADOOP-8689. We explain it in more detail as a showcase for the approach. Figure 3.3 shows a subset of changes making up the (erroneous) solution. A part of this patch provides separate trash cleanup intervals (`fs.trash.interval`) for client-side versus the server side. However, this change causes the test `TestHDFSServerPorts` to fail due to new call

---

<sup>3</sup>[http://hadoop.apache.org/issue\\_tracking.html](http://hadoop.apache.org/issue_tracking.html)

```

--- org/apache/hadoop/hdfs/server/namenode/NameNode.java
+++ org/apache/hadoop/hdfs/server/namenode/NameNode.java
@@ -511,9 +511,7 @@ public class NameNode {
private void startTrashEmptier( Configuration conf) throws IOException {
- long trashInterval = conf.getLong(
-   CommonConfigurationKeys.FS\ _TRASH\ _INTERVAL\ _KEY,
-   CommonConfigurationKeys.FS\ _TRASH\ _INTERVAL\ _DEFAULT);
+ long trashInterval = namesystem.getServerDefaults(). getTrashInterval ();
  if ( trashInterval == 0) {
    return;
  } else if ( trashInterval < 0){

```

Figure 3.3: Excerpt of code changes for issue Hadoop-8689 (simplified).

`getServerDefaults()` in the `startTrashEmptier()` function<sup>4</sup>. The failure of this test is reported in the bug report HDFS-3856.

In Step 1 of our approach, we retrieve a passing  $v_p$  as well as a failing  $v_f$  code version (passing or failing for test `TestHDFSServerPorts`). As shown in Table 3.2, the difference set  $Dif$  between  $v_p$  and  $v_f$  is very large, amounting to more than 109 kLOC.

Step 2 needs the failure stack trace of the test `TestHDFSServerPorts` to identify the failure site. As shown in Figure 3.2, code location `FSNamesystem.java:759` is the assumed failure site.

In Steps 3 and 4 we obtain the respective backward slices for  $v_p$  and  $v_f$  (922 and 759 JVM-bytecode statements), instrument the intersections  $IS_p$  and  $IS_f$  of  $Dif$  and backward slices (544 and 445 statements). In the subsequent Step 5 we obtain the code coverage of executing `TestHDFSServerPorts` on each of the two instrumented versions  $v_{p,inst}$  and  $v_{f,inst}$  (sizes 189 and 166 statements).

Finally, we can apply the filtering lemmas in Step 6 of Algorithm 1. It yields the final report for issue HADOOP-3856:

```

Suspicious failure –inducing changes:

In class: org.apache.hadoop.hdfs.server.namenode.NameNode:514

long trashInterval = namesystem.getServerDefaults(). getTrashInterval ();

```

<sup>4</sup>This explanation is taken from the comments on issue HDFS-3856 (Hadoop bug repository).

In the future, we will present to the developer not only this report but also the differences in code coverage to support failure understanding.

We also compared our suspect against the fix of this problem, which is committed in SVN revision 1377934. We found that indeed our hypothesis was correct. In the fix, the previously added faulty change is replaced with a new statement (shown in bold):

```

--- org/apache/hadoop/hdfs/server/namenode/NameNode.java
+++ org/apache/hadoop/hdfs/server/namenode/NameNode.java
@@ -511,13 +511,13 @@ public class NameNode {
private void startTrashEmptier( Configuration conf) throws IOException {
- long trashInterval =namesystem.getServerDefaults().getTrashInterval ()
+ long trashInterval = conf.getLong(FS_TRASH_INTERVAL_KEY,
+ FS_TRASH_INTERVAL_DEFAULT);
    if ( trashInterval == 0) {

```

### Issue HDFS-4887

The following case shows a limitation of our method but simultaneously points the way to extend it.

The considered issue is a failure in `TestNNThroughputBenchmark` test. The bug report states that the bug fix for the issue HDFS-4840 is responsible for the failure. The patch for fixing HDFS-4840 has introduced the following new (faulty) code to the `BlockManager` class of the HDFS codebase. The defect here comes from stopping the `ReplicationMonitor` while `NameNode` is still running.

```

if (!namesystem.isRunning()) {
    LOG.info("Stopping ReplicationMonitor.");
    if (!(t instanceof InterruptedException)) {
        LOG.info("ReplicationMonitor received an exception "
            + " while shutting down.", t);
    }
    break;
}
LOG.fatal("ReplicationMonitor thread received Runtime exception.", t);

```

### 3. Scalable Isolation of Failure-Inducing Changes

```
terminate(1, t);
```

Our approach applied to this case gives (after the filtering lemmas) an empty list of suspects, which shows a limitation of our method. The difficulty is caused by the small size of the set of instrumented statements (nine statements per version). Consequently, the coverage profiles have only two statements each (Table 3.2, middle rows). Both coverage profiles  $Cov_p$  and  $Cov_f$  contain the same statements, namely a `while` statement (line 3092 of `BlockManager` class) and `thread.sleep()` statement (line 3096 of the same class).

However, after investigating additional information provided by instrumentation (not used in this work) we discovered that the values of the conditional expression in the `while` statement are different in  $v_{p,inst}$  and  $v_{f,inst}$ . It is a natural extension of the current approach to consider such information. By extending the filtering lemmas, we can then include in the final report the conditional statements with diverging condition values.

Even assuming that the `while` statement would be included in the final report, it is not the precise root cause of the failure. However, this statement is within the method `run()` in the inner class of `ReplicationMonitor` in the `BlockManager` class. The method `run()` (about 20 LOC) indeed contains the defect. Thus, we can at least indicate the method containing the code for actual defect.

An investigation of the actual fix which is committed in SVN revision 1501841 shows that (because the defect is outside the changes), merely deleting the added changes from the new version does not solve the problem. To fix this bug, a check statement (shown in bold in following) was added to the conditional branch in the code:

```
--- org/apache/hadoop/hdfs/server/blockmanagement/BlockManager.java
+++ org/apache/hadoop/hdfs/server/blockmanagement/BlockManager.java
@@ -3129,6 +3138,9 @@
         + " while shutting down.", t);
     }
     break;
+ } else if (!checkNSRunning && t instanceof InterruptedException) {
+     LOG.info("Stopping ReplicationMonitor for testing.");
```

```

+   break;
}
LOG.fatal("ReplicationMonitor thread received Runtime exception.", t);
terminate(1, t);

```

### Issue HDFS-4282

Bug issue HDFS-4282 reports the failing of `TestEditLog.testFuzzSequence` test. Comments in the issue show that HADOOP-9103 breaks this test. Due to errors in decoding Unicode characters, in HADOOP-9103 a patch is created which has modified the code of UTF8 class in Hadoop Common project. However, this patch caused a failure of `TESTEditLog` test.

After applying our approach to this test case, the results point that the changes created in HADOOP-9103 are not faulty. However, the only difference in the execution profiles of passing and failing runs is a call of the method `toString()` in UTF8 class of the `io` package of Hadoop Common project. Also here Table 3.2 (bottom rows) give the sizes of code over all phases of our approach.

By investigating the execution profiles, we created a hypothesis that a solution to this bug is to add methods related to the `toString()` function. The real fix to this bug (committed in SVN revision 1418214) confirms our hypothesis. A new method `toStringChecked()` (with `IOException`) is added to the UTF8 class which now can throw an `IOException` for invalid UTF8 characters<sup>5</sup>.

### Issue Yarn-960

Bug reported in issue Yarn-960 manifests in the failure of tests `TestbinaryTokens` and `TestMRCredentials`. The reason for this test failure is the changes committed to the bug report YARN-701. Unfortunately, when applying our approach to this bug, we can not find the root cause of the error.

As we mentioned in the Section 3.2 (Step 2 in Algorithm 1), our approach needs a failure manifestation site in the failing version. However, in the stack trace of

<sup>5</sup><https://issues.apache.org/jira/browse/HDFS-4282>

### 3. Scalable Isolation of Failure-Inducing Changes

Table 3.2: Code size (#LOC or #JVM-bytecode statements) in different phases of our approach; *Dif* = difference set (in #LOC), *BSlice* = backward slice, *IS* = intersection set, *Cov* = coverage profile, Report = final report (in #LOC)

Issue	Version	Size				
		Diff #LOC	BSlice	IS	Cov	Report (#LOC)
HDFS-3856	passing	109207	922	544	189	1
	failing		759	445	166	
HDFS-4887	passing	1030	9	2	2	0
	failing		9	2	2	
HDFS-4282	passing	1325	795	367	88	1
	failing		800	372	89	

the failing executions for this issue, we can find only code locations in the third-party Java libraries and the JUnit framework. In our current experimental setting we cannot analyze these libraries (see the scenario description and assumptions in Section 3.4.1). However, our approach fails in this case, not due to a fundamental limitation but due to a (current) technical constraint that third-party artifacts like java libraries cannot be analyzed.

#### 3.4.2 Complexity of the Approach

RQ2 can be partially answered by inspecting Table 3.2. It shows the size of intermediate and final results in LOC (for *Dif* and Final Report) or JVM-bytecode statements. Note that in *Dif* a replaced line is counted twice - as an added and a removed line.

As an alternative to the Algorithm 1, we could have used the intermediate results for producing the final report. Specifically, this report could be based only on the code changes *Dif*, or only on the backward slice *BSlice*, or the intersection set *IS*, or on the coverage profiles *Cov*. For issues HDFS-3856 and HDFS-4282 executing all steps is the right way to achieve high specificity. Judging by these cases, our approach cannot be simplified.

However, for issue HDFS-4887, using any of the *BSlice*, *IS*, or *Cov* is feasible and could have led to pointing to the vicinity of the actual defect (Section 3.4.1). This result indicates that we could consider a dynamic workflow, where a result of

Table 3.3: Overheads of our approach compared to full instrumentation (Full instr.) and instrumenting only the code in *BSlice* (*BSlice* instr.); in “ $X/Y$ ”,  $X$  is the runtime slowdown (a factor) and  $Y$  size overhead of instrumenting (a factor); p = passing version, f = failing version, “-” = instrumentation not possible.

Issue	Ver.	Original Version		Full instr.	<i>BSlice</i> instr.	Our approach
		Runtime (s)	Size (MB)			
HDFS-3856	p	6	4.8	- / 4.60	1.20 / 1.04	1.00 / 1.00
	f	6	4.1	- / 5.40	1.00 / 1.02	1.00 / 1.00
HDFS-4887	p	9	4.8	1.60 / 4.60	1.00 / 1.00	1.00 / 1.00
	f	9	4.8	1.60 / 4.60	1.10 / 1.00	1.10 / 1.00
HDFS-4282	p	30	4.4	3.30 / 4.60	1.10 / 1.04	1.03 / 1.02
	f	30	4.4	3.00 / 4.60	1.03 / 1.04	1.00 / 1.02

an intermediate step is used directly for the final report if its size is below a certain threshold. Such workflow is subject to future work.

### 3.4.3 Performance Evaluation

To answer RQ3, we first evaluate Table 3.3. In a pair  $X / Y$ ,  $X$  represents the *runtime slowdown*, i.e., the ratio of time to execute the instrumented version divided by time to execute the non-instrumented version. Furthermore,  $Y$  is the size overhead of instrumenting, i.e., size of the instrumented version divided by the size of the original version.

The time and size overheads of the fully instrumented code (Full instr.) are significant. Code size increases by factor four to five, and execution time up to factor 3.3. Thus, full instrumentation is not efficient. However, instrumenting only the code in the backward slice (*BSlice* instr.) produces acceptable overheads. Even so, our approach beats the alternatives, having negligible overheads due to instrumentation.

Table 3.4 contrasts the running time of the failed test (Test time) against the total time needed to execute our approach (Total). As shown in the column “Total / Test”, the total time of our approach requires at most 3.3 times the duration of the failed test, and the latter is only one of many tests executed within a test suite.

Table 3.4: Running times of a failing test and times for various phases of our approach (times in seconds); Total / Test is the ratio of total approach time to test time

Issue	App. Time for Passing & Failing Version				Test Time	Total / Test
	Slicing	Instr.	Run	Total		
HDFS-3856	4	4	12	20	6	3.3
HDFS-4887	2	2	18	22	9	2.4
HDFS-4282	4	4	64	72	30	2.4

### 3.5 Chapter Summary

In this chapter, we introduced a scalable technique to localize the failure-inducing changes of functional bugs. Given a failing test and the source code of the previous correct and current buggy versions, we leveraged program analysis techniques to find the changes which were the root cause of the failure. However, we only evaluated our approach in four real cases, and the results are promising. Our approach found the exact failure-inducing change in two out of four cases. In one case, our approach could pinpoint the method containing the faulty code. In another case, our approach was unable to localize the faulty change as the stack trace of the crashing bug showed no path to the codebase of the application.

Our proposed approach differs from previous techniques. While Gupta et al. [46] focused on the changes in the input, our approach concentrates on the changes between the source codes of the two versions of a program. The presented approach also differs from delta debugging-based approaches [87, 137, 140]. Delta debugging narrows down the search space gradually by applying changes to the application iteratively. It requires a huge amount of test repetitions, causing a large time overhead. In our approach, we only execute the failing test. Our approach differs from SBFL techniques as we only use one test case. We also execute the program once. Therefore, there is only one failing and one passing run. This is different from SBFL approaches in which many failing and passing runs are used to rank the suspicious code statements.



## Chapter 4

# Automated Memory Leak Diagnosis via Version Comparison

Memory leaks are tedious to detect and require significant debugging effort to be reproduced and localized. In particular, many such bugs escape traditional testing processes used in software development. One of the reasons is that unit and integration tests run too short for leaks to manifest via memory bloat or degraded performance. Moreover, many of such defects are environment-sensitive and not triggered by a test suite. Consequently, leaks are frequently discovered in the production scenario causing high costs.

In this chapter, we propose an approach for the automated diagnosis of memory leaks during the development phase. Our technique uses the differences between software versions and existing test suites of the application. The key idea is to compare object (de-)allocation statistics (collected during unit/integration test executions) between a previous and the current software version. By grouping these statistics according to object creation sites, we can detect anomalies and pinpoint the potential root causes of memory leaks. Such diagnosis can be accomplished before visible memory bloat occurs, and in time proportional to the execution of test suite. We evaluate our approach using real leaks found in 7 Java applications. Results show that our approach has sufficient detection accuracy and is useful in isolating the leaky allocation site: exact defect locations rank relatively high in the lists of suspicious code locations if the tests trigger the leak pattern. Our

prototypical system imposes an acceptable instrumentation and execution overhead for possible memory leak detection even in large software projects.

## 4.1 Introduction

Software systems are becoming more complex due to growth in size and functionality. This increases the risk of *latent* defects which are difficult to be detected by the unit and integration testing. Memory leaks are the most prominent type of latent defects. They occur if objects remain in heap memory but are never reaccessed. However, also other types of latent defects (e.g., unterminated threads, unreleased file descriptors or pipes) finally lead to increased memory consumption and can be reduced to memory management problems.

Although the garbage collector is responsible for the memory management in memory-managed languages such as Java, C# or Python, they still suffer from memory leaks. The reason for this is that garbage collectors of these languages over-approximate object aliveness by its reachability [16]. Consequently, a reachable object is not disposed of even if it will not be used again. The most common scenario for such defects is forgotten references in collection data structures [132]. For example, objects encapsulating requests to a web server are frequently referenced from a collection data structure (list or map) which implements a processing queue. If the reference is not removed from the queue after the request is processed, the garbage collector cannot dispose of the associated object, and a memory depletion occurs.

Leaks are notoriously hard to detect, reproduce, and fix. One of the reasons is long latency between leak triggering and the manifestation of visible symptoms such as memory bloat or performance degradation [49]. A further problem is their sensitivity to inputs and execution environments [16]. As a consequence, many of such defects escape in-house quality assurance measures including unit, integration, and even performance testing. If these bugs be discovered in customer usage, they can have a significant economic impact. For example, a “latent memory leak bug” has caused a partial outage of Amazon’s EC2 cloud service on 22 October 2012 [5], affecting operations of hundreds of EC2 customers.

Memory leak diagnosis is an important problem for both researchers and practitioners. Based on an empirical study on the publications of the top tier software engineering conferences, the detection, and root cause analysis of memory leaks is among the top 10 highly rated research ideas [75]. Several tools [38, 49, 86] and research techniques are developed and designed to help developers to detect and isolate memory leaks. Most of these approaches for the diagnosis of memory leaks follow a “symptom to root cause” algorithm to detect memory issues [130]. One strategy is to apply staleness analysis to identify “dead” objects - those objects which can not be accessed for a long time [16, 50, 60, 93, 132]. Another group of works is based on analyzing heap growth [23, 59, 111, 112], or analysis of captured state [28, 86, 88, 129]. Most of these works assume that a leak has been already observed and test code triggering the leak is available. They help the developer with isolating the root causes of a leak at the cost of a proprietary execution environment (e.g., modified JVM [16]) or significant execution slowdown (e.g., 300-400% for Java [132]). Recent approaches for C/C++ focus on performance efficiency and promise a slowdown of  $\leq 3\%$  [60]. Such a low overhead makes them usable in a production environment and allows leak detection at customer sites.

However, none of these works address the fact that virtually all non-trivial software projects today (1) are developed as a series of relatively small code changes, and (2) are accompanied by an extensive suite of software tests which check (primarily) functional properties of the artifact. In this work we exploit (1) for an anomaly-detection based approach for leak diagnosis, and (2) for triggering memory leaks during in-house testing. Our approach supports the automated diagnosis of memory leaks during the software development phase and helps to pinpoint the root causes if a leak is captured. It requires only small modifications on software testing framework and no changes in the source code of tests and software. This is an essential factor for its acceptance and practicability in the context of existing projects. Since our method is an anomaly detection technique, it is not necessary to execute the test until significant memory bloat occurs (such bloat is a prerequisite for most existing methods). In this way, diagnosis time remains proportional to the time for executing the project’s test code.

### 4.1.1 Core Idea

Inspired by the Delta Debugging [138] for isolation of “crashing” errors we use software version comparison to uncover memory-related defects of the current (latest) software version. In this way, we can extract additional information which is not available when investigating each software version by itself an approach taken by previous work. Figure 4.1 outlines the approach. Given an older and current version of the software under development, we try to identify differences in memory allocation and deallocation behavior for each allocation site between these two versions. The workloads used here are unmodified (unit or integration) tests. We assign each such allocation site an anomaly score (denoted as *leak confidence*  $LC$ , Section 4.2.4) and rank the sites by this value. Unusually high  $LC$  values of top-ranked sites might indicate a new memory leak. Detection can be performed by manual evaluation of such top  $LC$  scores. For automated leak detection, the leak confidence of top-ranked sites can be compared against a threshold. If an alert is triggered, the ranking of sites supports debugging by indicating which allocation sites should be checked first.

### 4.1.2 Contributions

This chapter presents the following contributions:

- We propose an approach for diagnosis of memory leaks (Section 4.2.1) based on comparison of software versions under development. Contrary to other approaches, such diagnosis can be made during the development phase and before visible memory bloat occurs, in time proportional to the execution of a project’s test code.
- We validate our approach using both synthetic and real-world cases. We inject memory leaks at random places of different components of Apache Hadoop. Moreover, we evaluate our approach in seven real-world cases found in five medium to large-scale projects. We perform extensive empirical evaluations of the accuracy and efficiency of our approach and provide estimations on execution time and memory overheads (Section 4.4).

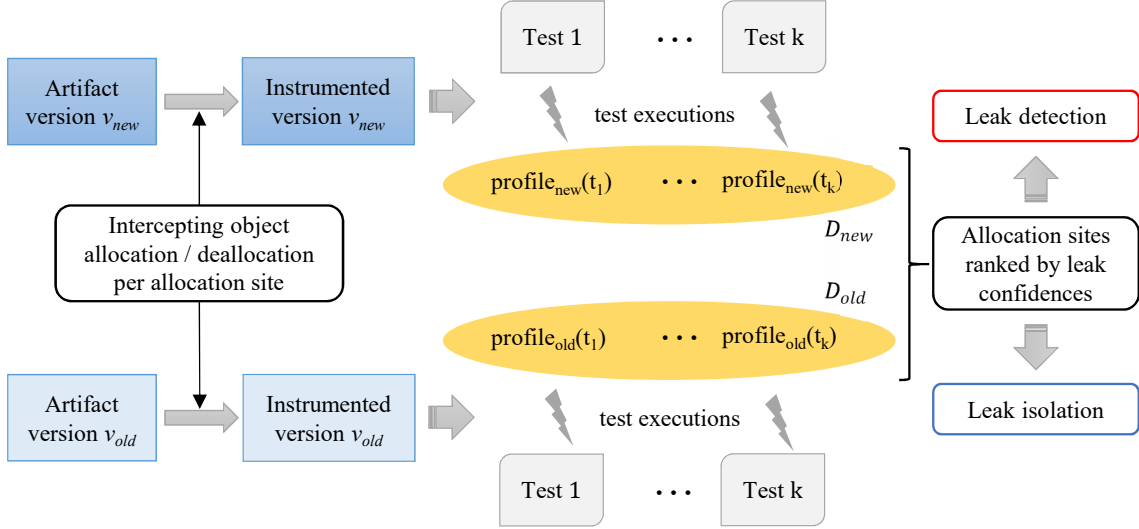


Figure 4.1: Overview of our approach.

- The results show that if the test code (provided with the project) triggers the execution of defect-related allocation sites, our approach can accurately diagnose leak-inducing allocation site with a low rate of false positives (Section 4.4). Furthermore, the overheads on our prototypical testbed indicate that the approach is feasible for performing leak diagnosis during the development phase and before the release time.

## 4.2 Leak Detection via Version Comparison

Our approach is based on comparing object allocation and deallocation behavior of a previous version  $v_{old}$  and a target (usually most recent) version  $v_{new}$  of a software artifact under development (Figure 4.1). Our method attempts to diagnose leaks which have been *newly introduced by code evolution between  $v_{old}$  and  $v_{new}$* . We do not assume that  $v_{old}$  is leak-free but leaks already present in  $v_{old}$  are less likely to be discovered. Thus, the choice of  $v_{old}$  should consider its reliability, mainly whether memory bloat has been observed during prolonged execution. Our approach works with both C/C++ and managed languages, with the only difference being technicalities of code instrumentation. The details of our prototypical implementation in Java are outlined in Section 4.3.1.1.

### 4.2.1 Approach Description

The first component of our method is monitoring all heap memory allocation and deallocation events. These events are grouped by individual *allocation sites*, i.e., code statements which triggered the allocation (for deallocation of memory/object, its allocation site is still the grouping criterion). To this end, we use code instrumentation via bytecode rewriting as described in Section 4.3.1.1. In the case of Java Virtual Machine (JVM) used for our prototypical implementation, all memory allocations on the heap are caused by object creation, and so we equal memory allocation and object instantiation on the heap.

An essential element of our approach is the comparison of allocation behavior at the granularity of *individual allocation sites* between different versions of the software under development. For each allocation site, its allocation profile under execution of multiple different tests is recorded. Metrics which exploit data from these multiple runs (Section 4.2.4) and also data comparison between software versions allow deciding about the potential presence of a leak. If yes, similar criteria determine the rank of a particular allocation site in a list of suspicious leak-inducing allocation sites (Section 4.2.5).

### 4.2.2 Instrumentation and Data Collection

Given software versions  $v_{old}$  and  $v_{new}$ , we identify (by static analysis) in each version all code locations which can allocate heap memory. In Java, such an allocation is triggered by object instantiation; in C/C++ this can also be caused by calling `new()` or related functions. We denote such a code location as an *allocation site as* and identify it by a source file ID, line number and (in case of Java) the class of the instantiated object. With such a specification, each allocation site uniquely corresponds to a location in the application bytecode.

In the subsequent phase of the diagnosis, we execute a series of software tests on instrumented versions of both  $v_{old}$  and  $v_{new}$  and collect allocation-related data from each test run. For clarity, we speak in the following about unit tests (symbol  $ut$ ), but in fact, any other type of test or terminating code can be used. In detail, for a given

unit test  $ut$  and software version  $v$  the following data is logged for each allocation site  $as$ :

- number  $n_a$  of objects allocated at  $as$  during the whole execution of  $ut$ .
- among all objects created at  $as$ , the number  $n_d$  of objects deallocated during the execution of  $ut$ .

Of particular interest is the number of *residual objects* which have not been deallocated upon termination (counted for a particular allocation site). Given an  $as$ , we compute the residual objects for  $as$  by  $n_r = n_a - n_d$ . Based on this number, we call  $as$  with  $n_r > 0$  as *suspicious* allocation site, otherwise as a *safe* allocation site.

The data obtained after running  $ut$  under artifact version  $v$  are the tuples  $(ID_{as}, n_a, n_d)$  for all allocation sites, where  $ID_{as}$  is data described above which uniquely identifies an allocation site. We call this set of tuples (over all allocation sites) an *allocation profile* and denote it by  $profile_{old}(ut)$  or just  $profile_{old}$  for  $v_{old}$  and by  $profile_{new}(ut)$  (or  $profile_{new}$ ) for  $v_{new}$ . Note that test execution is not always deterministic, and so the allocation profile might depend on a particular run.

The full results of the dynamic data collection are a set  $D_{old}$  of all allocation profiles (i.e., overall unit tests) executed under  $v_{old}$ , and an analogous set  $D_{new}$  for  $v_{new}$ .

### 4.2.3 Types of Allocation Sites

Some of the allocation sites present in  $profile_{old}$  do not exist or have not been executed in  $profile_{new}$  and vice versa. This gives rise to the following grouping of allocation sites:

- *Old allocation sites.* These are allocation sites which are recorded only in the allocation profile  $profile_{old}$ . As we are interested in leak discovery in the newer software version  $v_{new}$ , such allocation sites can be safely ignored.
- *New allocation sites.* These are allocation sites which are visible only in the new allocation profile  $profile_{new}$ .
- *Matching allocation sites.* These are allocation sites which appear in both old and new profiles.

#### 4. Automated Memory Leak Diagnosis via Version Comparison

Line insertions or deletions due to changes between previous and new software version change line numbering for all subsequent lines in the respective source file. This issue creates a technical problem for pairing (matching) allocation sites between versions. For example, if in source file  $F$  a new line after the line number  $k$  has been added, each line with number  $j > k$  in the older version of  $F$  corresponds to line with the number  $j + 1$  in the newer version of  $F$ . Since line numbers are part of data identifying an allocation site, the line numbers must be adjusted to identify all new and matching sites correctly.

We solve this problem with an algorithm, called *statement matching algorithm* which analyses the differences between source codes of  $v_{old}$  and  $v_{new}$  and adjusts line numbers in all profiles  $profile_{new}$ . The input of this algorithm are patches (`*.diff` - files) expressing code differences between  $v_{old}$  and  $v_{new}$  obtained by querying a software repository for the project. However, also any other data comparison tool which produces output in *unified diff* format can be used for preprocessing.

##### 4.2.3.1 Statement Matching Algorithm

As illustrated in Figure 4.2, our algorithm locates first the *drift candidates* (lines 5-9), i.e., the allocation sites which can be a matching site by adjusting the line number of that allocation site. To do this, we define three sets: old, new, and matching. If an allocation site is only in the older or newer version, we map it into the old or new set, respectively. If an allocation site is in both versions with the same source line number and class name, we map it to the matching set.

For older set entries, our approach attempts to compute drift amount pointing to the new source line in the newer version (i.e., an entry in the new set). For each entry in the drift candidates list, we check whether the source line of that entry has been moved or removed after the changes in the newer version. If it is removed, we also remove that entry from the drift candidates list (lines 13-17). Otherwise, we calculate the drift (lines 18-28). The drift is yielded by the sum of the added lines reduced by the sum of the removed lines if the addition or deletion of a line occurs before the line number of the drift candidate in the same source file. Finally, after updating the drift candidates from the old set (lines 30-31), we try to find an entry in the new set for each of the remaining drift candidates. Each entry in the drift candidates which



**Auxiliary functions:**

- *oldAS*, *newAS*: allocation sites pairs for older and newer versions
- *lineType(lineNum)*: returns type of the source line number of an allocation site in the diff file, *deleted* for removed line and *added* for added line
- *labeledOldAS*, *labeledNewAS*: Two lists of allocation sites with match labels for each allocation site in *oldAS* and *newAS*. For each allocation sites, *matchLabel* = 0 if it exists in both *oldAS* and *newAS*, “-2” if it only exists in *oldAS* and “-1” if it only belongs to *newAS*

**Input:** *diffFile*, *labeledOldAS* as a set of allocation sites only in *oldAS*

**Output:** amount of drifts for allocation sites in *oldAS*

**function** computeDrift(*labeledOldAS*, *oldVersion*, *newVersion*):

```

1: updatedOldAS, driftCandidates, chunkList ← new List[]
2: diffFile ← getDifferenceUsingDiff(oldVersion, newVersion)
3: chunkList ← parseDiffToChunksOfChanges(diffFile)
5: for allocation site in labeledOldAS do
6:   if matchLabel(allocation site) == -2 then
7:     driftCandidates ← allocation site
8:   end if
9: end do
10: for allocation site in driftCandidates do
11:   srcFile, lineNum ← getSiteParam(allocation site):
12:   for chunk in chunkList do
13:     if (srcFile and lineNum) in chunk then
15:       if lineType(lineNum) == deleted then
16:         break
17:       end if
18:       initialize drift variable
19:       chunkStartLineNum ← getChunkLineNum(chunk)
20:       if lineNum > chunkStartLineNum then
21:         for statement in chunk do
22:           if lineType(statement) == deleted then
23:             drift ← drift - 1
24:           else if lineType(statement) = added then
25:             drift ← drift + 1
26:           end if
27:         end do
28:       end if
28:     end if
29:   end do
30:   lineNum ← lineNum + drift
31:   updatedOldAS ← (srcFile:lineNum, class)
33: end do
34: return updatedOldAS

```

Figure 4.2: Matching algorithm for drift computation of two software versions.

can be matched to an entry in the newer set using drift algorithm will be removed from both older and newer sets and mapped to the matching set.

#### 4.2.4 Leak Confidence Score

This section describes the computation of a scalar anomaly measure called *leak confidence score* ( $LC$ ) from the complete sets of allocation profiles  $D_{old}$  and  $D_{new}$ . This score maps each allocation site  $as$  encountered in  $v_{new}$  to a numerical value  $LC(as)$  in  $[0, 1]$ , with higher values indicating higher defect probability. The general form of  $LC(as)$  is:

$$LC(as) = A(as) * B(as) * C(as), \quad (4.1)$$

with terms  $A(as)$ ,  $B(as)$ , and  $C(as)$  described below. Their definitions are based on our empirical observations and our prior research on memory leak detection [65].

To simplify the notation, we introduce for  $x \neq 0$ ,  $y \neq 0$  the *normalized harmonic mean of  $x$  and  $y$*   $H(x, y)$  defined by:

$$H[x, y] = \frac{1}{1/x + 1/y}.$$

We set  $H = 0$  if  $x = 0$  or  $y = 0$ .

##### 4.2.4.1 Factor $A(as)$

This factor captures the overall strength of an allocation site  $as$  in terms of residual objects. It only considers the version  $v_{new}$ . It exploits as a core idea the observation that a leaky allocation site  $as$  is likely to deallocate only a few of its allocated objects. This should hold for any unit test  $ut$  and will yield a high “relative” number of residual objects  $n_r(as, ut)/n_a(as, ut)$ .

We can achieve a higher robustness if we consider the set  $UT(as)$  of all unit tests which cover  $as$ . This motivates the definition of the *rate of residuals*  $ResidR$ : it is

the fraction of allocated objects which are not deallocated during the execution of relevant unit tests:

$$ResidR(as) = \frac{\sum_{ut \in UT(as)} n_r(as, ut)}{\sum_{ut \in UT(as)} n_a(as, ut)}. \quad (4.2)$$

Our experiments have shown that leaky allocation sites have higher *absolute* number of residual objects  $\sum_{ut \in UT(as)} n_r(as, ut)$  (summed over all relevant unit tests). The final formula for  $A(as)$  combines both expressions via the normalized harmonic mean:

$$A(as) = H[ResidR(as, UT), \sum_{ut \in UT} n_r(as, ut)]. \quad (4.3)$$

#### 4.2.4.2 Factor $B(as)$

This factor captures how easily a memory leak is triggered at an allocation site  $as$  by a unit test that exercises it. It only considers version  $v_{new}$ . If an allocation site  $as$  becomes a leak cause, then it is likely to create residual objects under many different execution patterns (represented by different unit tests). We define (*"dirty"*) *test rate*  $TestR(as)$  as the fraction of unit tests  $ut$  (among unit tests in  $UT(as)$ ) for which the number of residual objects  $n_r(as, ut)$  is greater than zero.

The metric  $TestR(as)$  can be inaccurate in case of small  $|UT(as)|$ , i.e., if allocation site  $as$  executed only few times. To dampen the impact of such cases, we use harmonic mean of  $TestR(as)$  and  $|UT(as)|$ :

$$B(as) = H[TestR(as), |UT(as)|]. \quad (4.4)$$

#### 4.2.4.3 Factor $C(as)$

This factor measures the “leakiness” of an allocation site  $as$  in the new version  $v_{new}$  compared to the old version  $v_{old}$  (and hence considers both versions). If an allocation site  $as$  becomes a leak cause due to evolution between versions  $v_{old}$  and  $v_{new}$ , the number of its residual objects  $n_r(as)$  is likely to increase in  $v_{new}$ . We define the  $NresidChR(as)$  as the relative change in the number of residual objects of  $as$  in the older version to the newer version:

#### 4. Automated Memory Leak Diagnosis via Version Comparison

$$NresidChR(as) = \frac{n_r(as, v_{new}) - n_r(as, v_{old})}{n_r(as, v_{new})}.$$

Note that for new allocation sites,  $n_r(as, v_{old})$  is equal to zero, and so we have  $NresidChR(as) = 1$  in such cases.

Allocation sites with larger value of numerator  $\Delta(as) := n_r(as, v_{new}) - n_r(as, v_{old})$  have higher probability to be a memory leak. Therefore we define  $C(as)$  as the harmonic mean of  $NresidChR(as)$  and  $\Delta(as)$ :

$$C(as) = H[NresidChR(as), \Delta(as)]. \quad (4.5)$$

##### 4.2.5 Ranking

The sets  $D_{old}$  and  $D_{new}$  of all allocation profiles are used to compute the *Leak Confidence LC* (Section 4.2.4) for each allocation site triggered in the current software version  $v_{new}$ . In the next analysis step, the allocation sites in  $v_{new}$  are ranked by their *LC* values in decreasing order. In this way, we obtain a *ranked list of suspects* with most suspicious sites being top-ranked.

##### 4.2.6 Discussion

**Why unit testing?** Software systems are becoming more complicated due to growth in size and functionality. This growing complexity causes more bugs in software. Fixing bugs during the development phase can save a lot of time and costs. Unit testing is a simple and efficient way to find most of the bugs during the development phase. Many of today's software projects have included an extensive tests suite. However, unit testing is usually used for testing the functional properties. Due to the severity of memory leaks, it would be highly beneficial to find such bugs before a software release. For this reason, in this work, we tried to propose an approach which uses unit testing to pinpoint memory leaks in the development phase.

**Fixing memory leaks.** Although the goal of our approach is to isolate the leak-inducing allocation sites, it is the main and challenging step toward fixing the memory

leak. During the search for real-world memory leaks in the bug repositories of Java applications, we noticed that many reported leaks are not a real memory leak. Many leak-related issues are labeled as “invalid” or “not a problem”. Limited knowledge about the behavior of memory leaks or interpreting other problems such as race condition as a memory leak are some of the reasons for having an incorrect report of memory leak. This confirms that the proper isolation and detection of memory leaks acts as the first and main step toward fixing and removing memory leaks.

## 4.3 Experimental Design

### 4.3.1 Methodology

To evaluate how our approach works in the diagnosis of memory leak, we used both synthetic and real-world memory leaks. All of the experiments are conducted in a virtual machine with 8 GB physical memory running on a 2.9 GHz Intel Dual Core i7-3520M CPU with Ubuntu 12.04. We used Java 1.6.0\_27 with 4 GB heap size. Framework, data and evaluation results of our approach are available online<sup>1</sup>. Note that all of the steps in order to compute the leak confidence analysis and to report the ranked list of the suspicious allocation sites are fully automated.

#### 4.3.1.1 Instrumentation

We use instrumentation to record the allocation profiles (Section 4.2.2). In Java, an allocation site corresponds to a unique location in the bytecode. We specify it by the name of the corresponding source file, the line number of this allocation site in the source file, and the class of instantiated object. In this way, the developer can identify (during leak isolation) the code location more quickly than via bytecode position.

**Object allocation.** To monitor and record object allocations we use the library *java-allocation-instrumenter* [79]. It performs static code analysis and instruments bytecode at each allocation site. This instrumentation calls our proprietary code hook which inspects the current stack trace. We retrieve the code location of the caller (i.e., allocation site source file and line number) and save this information together with

---

<sup>1</sup><http://1drv.ms/1GU3Pfn>

Table 4.1: Subject programs. Column “# Unit Tests” shows the number of unit tests used in the evaluation.

Subject Program	# LOC	# Unit Tests
Hadoop-Common	94k	234
Hadoop-Yarn	163k	85
Hadoop-HDFS	200k	315
Hadoop-MapReduce	157k	166
Snappy-Java	2.5k	6
Apache Thrift	6k	18
Apache Solr	38k	19
Apache Nutch	27k	31

the class of the instantiated object in a hash map. Our hook also checks whether the allocation site is located in one of the source files of interest. Note that we exclude code in the third-party libraries.

**Object deallocation.** The final function of the code hook is to prepare notifications of object deallocations. To this end we link via *phantom references* (using Java’s *sun.misc.Cleaner.create()* method) each newly allocated object with a proprietary callback method. This method executes precisely once after the object has been deallocated. It can identify the allocation site *as* for the object and updates the deallocation count for it.

After a test is finished but the JVM is still alive, we enforce a garbage collection and record the statistics  $n_a$  and  $n_d$  for each monitored allocation site.

#### 4.3.1.2 Experimental Setup

We evaluated the accuracy of our approach on two sets of known memory leaks: (a) Eight synthetic defects causing controllable memory leaks injected in the source code of the Apache Hadoop framework; (b) Seven real leaks found in different Java applications/projects. The subject programs used in the evaluation are listed in Table 4.1.

For experiments on the dataset (a) we injected defective code into the four components of Apache Hadoop<sup>2</sup>, namely Common, Yarn, HDFS and MapReduce. Apache Hadoop is a large-scale open source software project containing more than 1

---

<sup>2</sup><http://hadoop.apache.org>

Table 4.2: Hadoop source code versions used in the evaluation of synthetic leaks. Column “Development Revision” shows the used revisions. Column “Changed Files” shows the differences between  $V_0$  and  $V_1$  in terms of files, where “m” and “a” indicate the number of modified and added files, respectively. Column “#Changed Lines” states the total number of changed lines between both versions.

Version	Development Revision	Changed Files	#Changed Lines
$V_0$	r1446308	42m + 1a	1401
$V_1$	r1450807		

million lines<sup>3</sup> of Java code with many development revisions. Each component includes a comprehensive test suite. These features make Apache Hadoop a suitable environment for the evaluation of our approach for suitability for large-scale software systems.

In the experiments using dataset (b), we collected seven real leaks from five large open source applications. Three out of seven cases are from Apache Hadoop. The rest is collected from four other applications: Snappy-Java, Apache Solr, Apache Nutch, and Apache Thrift (Table 4.1). The first column of Table 4.5 shows the details of the corresponding leaks.

Snappy-Java<sup>4</sup> is a port of Snappy project used in many programs and frameworks such as Apache Spark, Big Table, MapReduce for compression and decompression. Apache Solr<sup>5</sup> is an open source enterprise search platform mainly for full-text searching. Apache Nutch<sup>6</sup> is an open source, scalable, feature-rich web search engine. Apache Thrift<sup>7</sup> is a software framework for the development of scalable and efficient cross-language services.

#### 4.3.1.3 Implementation of Synthetic Defects

To simulate both matching and new allocation sites, we inject code triggering synthetic memory leaks into each of the two consecutive Hadoop development revisions  $V_0$  and  $V_1$  described in Table 4.2. For each of  $V_0$  and  $V_1$ , the leak-triggering

<sup>3</sup>On February 18, 2016, Open hub (<https://www.openhub.net/p/Hadoop>) was reporting that Apache Hadoop has 1,107,731 lines of Java code

<sup>4</sup><https://github.com/xerial/snappy-java>

<sup>5</sup><http://lucene.apache.org/solr>

<sup>6</sup><http://nutch.apache.org>

<sup>7</sup><https://thrift.apache.org>

#### 4. Automated Memory Leak Diagnosis via Version Comparison

**Auxiliary data structure:** static array *leakingObject*  
private final static java.util.Collection <byte[]> leakingObject =  
new java.util.LinkedList <byte[]>();  
**function** addLeak (*allocationSiteIndex*, *AASType*, *leakStrength*) :  
1: **if** *AASType*  $\neq$  *invisible* **then**  
2:   leakSize = random integer from  $1, \dots, \text{leakStrength}$   
3:   byte[] allocatedMemory = new byte[leakSize];  
4:   **if** *AASType* = *leaky* **then**  
5:     leakingObject.add(allocatedMemory);  
6:   **end if**  
7: **end if**

Figure 4.3: Pseudo code of the Artificial Allocation Site (AAS) used as a leak-triggering defect.

code is inserted at matching code locations. These consequent development revisions were randomly selected from the Hadoop code repository. The eight code locations for injections were also chosen randomly (yet we covered all four Hadoop components).

Each leak-triggering patch is implemented as a piece of code which creates a byte array and can add it to a static Java collection each time the code is executed. Figure 4.3 shows the pseudo code of such an injection. We call the Java statement creating a byte array an *artificial allocation site* (AAS).

We can control the settings of each AAS via parameter *AAS type* to enforce three states: *invisible*, *non-leaky* and *leaky*. An AAS is invisible if no byte array is allocated upon its execution. We use this state to simulate allocation sites of type *new* in  $V_1$ : this is achieved by setting AAS to invisible in  $V_0$  and its sibling in  $V_1$  to non-leaky or leaky. If an AAS in  $V_0$  and its sibling in  $V_1$  are both either non-leaky or leaky, the AAS in  $V_1$  is obviously of type *matching*.

Furthermore, we control another parameter called *leak strength*  $s$  (in bytes) which bounds the maximum size of a created synthetic leak (i.e., allocated byte array). The actual amount of memory allocated for each leak is picked randomly from interval  $[1, \dots, s]$  using uniform distribution. Using this randomness property, we ensure that our injected leak is non-deterministic and therefore it is more similar to the real-world cases.



However, large values of  $s$  cause to inflate the accuracy of our leak diagnosis approach. Identifying a large memory leak in comparison with other allocation sites with small or no residual objects can be “too easy” even with an imprecise leak detector. Therefore we use as the *leak strength the value*  $s = 10$  (bytes) for matching sites, and *leak strength*  $s = 2$  for simulated new allocation sites. These small values are conservative and selected to evaluate whether our approach is accurate even in case of small leaks.

### 4.3.2 Research Questions

To evaluate our approach, we designed experiments to answer the following research questions:

**RQ1:** *How accurate is our approach in diagnosing memory leaks caused by synthetic defects (dataset a)?*

To answer this question, we activate each of the 8 synthetic defects (one by one) to cause memory leaks, in each case with allocation site types *new* and *matching*. In each of the resulting 16 cases we apply our approach and report: 1) the leak confidence score for the injected (artificial) allocation sites and 2) the the position of the injected allocation site in the ranked list of the suspicious allocation sites (Section 4.4.1).

**RQ2:** *What is the accuracy of our approach for diagnosis of real memory leaks (dataset b)?*

To answer this research question, we perform the leak confidence analysis for each of the seven issues listed in Table 4.4 on two application variants: on a leaky version (i.e., containing memory leak), and a non-leaky version (an application version committed to the repository before the leaky version). We report: 1) the leak the confidence score for the leaky allocation site, 2) the position of this allocation site in the ranked list of the suspicious allocation sites, 3) the number of allocation sites with the  $LC > 0$ , and the value of  $LC_{max}$  (Section 4.4.2). Also, we describe the details for each of these seven issues.

**RQ3:** *What is the impact of each factor of the leak confidence metric  $LC$  on the accuracy of memory leak diagnosis?*

In Section 4.2.4, we proposed three factors to compute the leak confidence metric. This research question evaluates the impact of these factors on the performance of

our approach on dataset b. To answer RQ3, we perform the leak confidence analysis separately for each factor (over all of the subject programs), and then compare the results with the results obtained by the last metric (Equation 4.1).

**RQ4: *What is the overhead of our approach in terms of runtime and memory usage?***

To answer RQ4, we evaluate our approach in terms of runtime and memory overhead (Section 4.4.4). For each unit test we collect metrics *Runtime* and *Resident Set Size* (RSS) after the execution of each unit test. Finally, we aggregate the collected measures overall unit tests for the program in question and report the overall results for each case.

## 4.4 Experimental Evaluation

In this section, we answer the research questions.

### 4.4.1 Experiment I: Evaluation of Synthetic Defects

In the first experiment, we try to answer RQ1 by evaluating the accuracy of our approach using synthetic defects. To this end, we designed two scenarios: new site analysis and matching site analysis (Section 4.2.3).

**New site analysis.** Memory leak is introduced in the newer version of the software in question. Given an allocation site  $as$  and a consecutive version pair  $\langle V_0, V_1 \rangle$ ,  $as$  only exists in the heap profile of version  $V_1$  with  $n_r(as)_{v_1} > 0$ . For this scenario, we insert the synthetic leak (pseudo code shown in Figure 4.3) in the random places of version  $V_1$  of Hadoop program shown in Table 4.2. Then we adjust leak strength  $s = 2$  for each leak (i.e., injected leak has a size of at most 2 bytes).

**Matching site analysis.** Leak-inducing allocation site already exists in the older version, but it is inactive and is triggered by the committed changes in the newer version. Scenario “b” means that  $as$  exists in the heap profile of both versions with  $n_r(as)_{v_1} > n_r(as)_{v_0}$ . In this scenario, we insert the synthetic leak in the random places of both versions of Hadoop shown in Table 4.2. Then we adjust leak strength  $s = 0$  for each leak (i.e., leaky allocation site is visible but with a very small size) in

Table 4.3: Evaluation results of synthetic memory leaks. Column “Leak ID” indicates each synthetic leak. Section (a) reports the  $LC$  value for the leaky AAS (Column “ $LC$ ”) and also the difference between the  $LC$  values of the first two entries of the ranked list (Column “ $LC$  diff”). Section (b) shows the result of leak isolation: Column “rank” reports the rank of leaky AAS in the ranked list and Column “#Candidates” shows the number of allocation sites with  $LC > 0$ .

Leak ID	AAS	(a) $LC$ Analysis		(b) Leak Isolation	
		$LC$	$LC$ Diff	Rank	# Candidates $LC > 0$
common#1	new	0.99	0.10	1	86
common#2	matching	0.99	0.33	1	5161
yarn#1	new	0.95	0.08	1	121
yarn#2	matching	0.96	0.3	1	5572
yarn#3	new	0.96	0.09	1	97
yarn#4	matching	0.96	0.25	1	5838
yarn#5	new	0.96	0.09	1	178
yarn#6	matching	0.96	0.23	1	5218
hdfs#1	new	0.99	0.11	1	129
hdfs#2	matching	0.99	0.24	1	5265
hdfs#3	new	0.99	0.13	1	120
hdfs#4	matching	0.99	0.26	1	5265
mr#1	new	0.97	0.1	1	190
mr#2	matching	0.97	0.12	1	5732
mr#3	new	0.94	0.09	1	170
mr#4	matching	0.94	0.14	1	5492
no leak	new	0.88	0.06	-	156
no leak	matching	0.69	0	-	5974

the version  $V_0$  and leak strength  $s = 10$  for each leak (i.e., injected leak has size of at most 10 bytes) in the version  $V_1$ .

To find the suspicious allocation site in each scenario, we execute all unit tests over both old and newer versions and collect the heap profiles overall unit tests (Section 4.2.2). Finally, we use our leak confidence analysis (Section 4.2.4) to identify the most suspicious allocation sites and report these allocation sites as a ranked list to the developer. All experimental data for synthetic defect evaluation of the new and matching sites are reported in the Table 4.3. The results show that our approach can diagnose the leak-inducing allocation sites accurately with a high leak confidence score.

#### 4. Automated Memory Leak Diagnosis via Version Comparison

Section (a) in the Table 4.3 reports the highest leak confidence value in each case in addition to the difference between the leak confidence value of the first two entries of the ranked list. Although the size of the injected leaks was small, our approach could detect all of them with a high leak confidence. In all of these cases, the injected leaks had the highest leak confidence among all of the other reported allocation sites in the ranked list. This is an essential feature of our approach - it can report the leak-inducing allocation site with a high leak confidence score. Also, the results show that in most of the matching memory leaks, there is a relatively big difference between the leak confidence value of the first two ranked allocation sites. It reveals that if a leak pattern is triggered due to changes in the newer version of the code, then our approach can detect it. However, for new memory leaks there exists a small gap between the first two entries in the ranked list.

With further investigation of the experimental data, we found out that if an allocation site is leaky, the number of residual objects in most of the unit tests exercising this specific allocation site is greater than zero. Furthermore, if an allocation site with residual objects greater than zero is triggered frequently, our approach reports it with a high leak confidence value. This makes sense in the real-world scenarios because some functions are more frequently executed and memory issues due to the execution of these functions cause more effects in the performance of the application (rather than a memory defect in a rarely executed function).

Table 4.3(b) reports the results of leak isolation. There are a large number of allocation sites with several residual objects greater than zero which can be observed by monitoring the running unit tests. Having a ranked list on these allocation sites to highlight the most-suspicious ones can accelerate finding the root cause of memory leaks by developers. To this end, our approach reports a ranked list based on the leak confidence analysis (Section 4.2.5) for new and matching sites. The results show that all synthetic memory leaks are ranked first in both types of allocation sites.

#### 4.4.2 Answer to RQ2: Evaluation of Real-World Issues

The second experiments evaluate the efficiency of our leak diagnosis approach in real-world cases. We collected seven real cases from different Java projects listed in the

Table 4.4: Information related to the real memory leaks. Column “#Trig. UT(#Total UT)” shows the number of unit tests which trigger the leak pattern. The number in parenthesis indicates the total number of unit tests for that project.

Issue	Status	Leaky Version	Non-Leaky Version	#Trig. UT (#Total UT)
hadoop-8632	fixed	2.0.0a	0.20.0	135(234)
hdfs-5671	fixed	2.2.0	2.0.6-alpha	2(315)
yarn-1382	fixed	2.2.0	0.23.11	23(46)
thrift-1468	fixed	0.5.0	0.4.0	0(18)
snappy-91	fixed	1.1.1.5	1.1.1.3	2(6)
solr-1042	fixed	1.3	1.2.1	11(19)
nutch-925	fixed	1.2	0.8	13(31)

Table 4.4. For each of these cases, the developers fixed the reported leak by applying new patches to the leaky version. We used these patches to find the leak-inducing allocation site and also to verify the accuracy and efficiency of our approach. Note that to fix memory leaks, developers have changed multiple lines of code in different files. However, in each case, we marked the leak-inducing allocation site as the root cause of the memory leak.

For each real case, we reproduced the leaky version from the information provided by the issue report in the bug repository. For non-leaky versions, we manually found a non-leaky version *prior* to the leaky one by comparing the source codes. In this way, we obtained seven pairs of software versions (one per issue):  $\langle non\text{-}leaky\ older\ version, leaky\ newer\ version \rangle$ . Although the manual finding a non-leaky version is quite tedious, this step is needed only in the evaluation, and it is not a limitation of our approach.

For experiments, we collect and analyze data before and after leak-inducing changes in each of the seven cases. For each case, we executed all unit tests for both non-leaky and leaky versions of the program in question. Then we applied the leak confidence analysis to find the leak-inducing allocation site corresponding to a memory leak.

Table 4.5 shows the result of leak isolation and leak confidence analysis for each case. In 4 out of 7 cases, the allocation site contributing to memory leaks were ranked in the top 10. In issue Snappy-91, we report the instantiation site of the leaky

#### 4. Automated Memory Leak Diagnosis via Version Comparison

Table 4.5: Results of leak confidence analysis and leak isolation for real cases. Section (a) shows the result of the leak isolation: rank of the leak-inducing allocation site and the size of the ranked list of suspects with  $LC > 0$ . Section (b) reports as  $LC$  the leak confidence score for the leak-inducing allocation site and as  $LC_{max}$  the largest leak confidence value among all sites in the ranked list.

Issue	(a) Leak Isolation		(b) $LC$ Analysis	
	Rank	#Candidates ( $LC > 0$ )	$LC$	$LC_{max}$
hadoop-8632	2	3007	0.98	0.99
hdfs-5671	668	5524	0.54	0.99
yarn-1382	115	2368	0.79	0.96
thrift-1468	-	151	-	0.71
snappy-91	1	40	0.57	0.57
solr-1042	4	129	0.77	0.83
nutch-925	8	226	0.8	0.91

allocation site as rank one. The root cause of memory leaks in issues HADOOP-8632, Solr-1042 and Nutch-925 were ranked 2, 4 and 8, respectively.

For issues HDFS-5671 and YARN-1382, our approach assigns low ranks to the leak-inducing allocation sites. The reasons are discussed in Section 4.4.2.1. In case of the issue Thrift-1468, our approach did not include the root cause of memory leak in the list of suspects. This can be attributed to the fact that the unit tests at all do not trigger the leak-activating allocation site. Consequently, this allocation site did not appear in the list of known allocation sites. In the following, we provide a detailed analysis of each real case.

##### 4.4.2.1 Case Studies

###### Hadoop Common

Issue HADOOP-8632<sup>8</sup> reports that a newly introduced variable `CACHE_CLASSES` in the `configuration` class caused a leak on the class loaders. This variable is a part of a patch for solving a performance regression bug in issue HADOOP-6133. Therefore the changes in the HADOOP-6133 could be the root cause of this memory leak. However, the memory leak was reported three years after HADOOP-6133 was submitted in issue

<sup>8</sup><https://issues.apache.org/jira/browse/HADOOP-8632>

HADOOP-8632. Consequently, a developer has modified the code with converting the strong reference of the class to its *ClassLoader* to a weak reference:

```

--- org/apache/hadoop/conf/Configuration.java
+++ org/apache/hadoop/conf/Configuration.java

@@ -219,8 +220,8 @@
- private static final Map<ClassLoader, Map<String,
-   Class<?>>> CACHE_CLASSES = new
-   WeakHashMap<ClassLoader, Map<String, Class<?>>>>();
+ private static final Map<ClassLoader, Map<String,
+   WeakReference<Class<?>>>> CACHE_CLASSES = new
+   WeakHashMap<ClassLoader, Map<String, WeakReference<Class<?>>>>();

```

Although there is a large number of changes between the leaky and non-leaky version, our leak confidence analysis could pinpoint the instantiation site of the variable *CACHE\_CLASSES* correctly with a high leak confidence score, placing it at position two in the ranked list of suspects. The reason for such a high leak confidence score is that the leak pattern, in this case, is exercised by many unit tests. Also, the residual objects for the leak-inducing allocation site were greater than zero in all unit tests executing this site.

## Hadoop HDFS

Issue HDFS-5671<sup>9</sup> reports a leak in the *getBlockReader* method of the *DFSInputStream* class. When a client requests a file's block to *DataNode*, the *BlockReader* will be called from *DFSInputStream* class. If the cache is missing, a new pair for *BlockReader* will be created. However, if an *IOException* is thrown during creation of a new *BlockReader* with the given pair, then the TCP socket used by the *regionserver* will not be closed. This causes too many *close-wait* status which is a socket (connection) leak and finally results in a huge memory footprint. To solve this issue, developers changed *DFSInputStream* class and moved the statement which creates a new *BlockReader* into a try block which closes the peer when there is no new *BlockReader*.

<sup>9</sup><https://issues.apache.org/jira/browse/HDFS-5671>

```
Peer peer = newTcpPeer(dnAddr);
- return BlockReaderFactory.newBlockReader(
-     dfsClient .getConf(), file , block, blockToken,
-     startOffset , len , verifyChecksum, clientName, peer ,
-     chosenNode, dsFactory, peerCache,
-     fileInputStreamCache, false , curCachingStrategy);
+ try {
+     reader = BlockReaderFactory.newBlockReader(
+         dfsClient .getConf(), file , block, blockToken, startOffset ,
+         ...
+     } finally {
+         if (reader == null) {
+             IOUtils . closeQuietly (peer);
+         }
+     }}
```

We applied our approach to pinpoint the site which calls the new *BlockReader*. We used the version reported by the issue as the leaky version. However, for the correct version, we chose a much earlier version before the leaky one because issue HDFS-5671 was not reported as a regression error. We exercised this leak to check the reaction of our approach to the enormous amount of changes between the two software versions (i.e., leaky and non-leaky versions).

Our approach reported the suspicious statement, however with a low leak confidence score and low ranking position. The main reason for this low accuracy is the low number of unit tests which trigger a memory leak. The leak confidence value is directly dependent on the number of unit tests which trigger a memory leak pattern. The more triggering unit tests, the higher the value of the leak confidence. However, in this case, in contrary to issue HADOOP-8632, only two unit tests (from the total number of 315 unit tests) exercised memory leak pattern which contributed to low accuracy.



## Hadoop YARN

Issue YARN-1382<sup>10</sup> reports that *NodeListManager* class in *Hadoop-YARN* contains a memory leak when a node in the unusable nodes set never comes back. This issue was reported in the comments of another issue (YARN-1343). Based on the description of the issue, although this is not a huge memory leak, it can be accumulated if the *NodeManager* are configured with ephemeral ports which assumes that the nodes are still new when they are released.

We applied our approach to check whether it can find the leak-inducing allocation site for this memory leak. Our approach could pinpoint the leaky site with a high leak confidence score. Also, developer fixed this issue mainly with removing this allocation site (and also its corresponding code pieces) which is not used in the code anymore:

```
– private Set<RMNode> unusableRMNodesConcurrentSet = Collections
–   .newSetFromMap(new ConcurrentHashMap<RMNode,Boolean>());
```

Despite the high *LC* value for the leak-inducing allocation site, its position is low in the ranked list. The main reason for the low accuracy is a large number of changes between the leaky and non-leaky versions. Our approach requires a pair of consecutive versions of the application in order to categorize the allocation sites and to perform the leak isolation using the leak confidence analysis. If these two versions have a huge source code difference (i.e., many versions between the prior and current versions), the total number of newly introduced allocation sites increases substantially. This considerably affects the rank of the leak-inducing allocation site and also increases the number of potential leak suspects.

## Snappy-Java

Snappy-Java suffered from a severe memory leak after updating from version 1.1.1.3 to the version 1.1.1.4. This memory leak was reported in issue Snappy-91<sup>11</sup>. It had negative effects on Apache Spark 1.2.0 which updated the Snappy-Java library to the version 1.1.1.4. Due to memory leak introduced in the newer version of the Snappy-Java, Spark developers roll-backed to the previous version of the

<sup>10</sup><https://issues.apache.org/jira/browse/YARN-1382>

<sup>11</sup><https://github.com/xerial/snappy-java/issues/91>

```
+ inputBuffer = inputBufferAllocator . allocate ( inputSize );  
+ outputBuffer = inputBufferAllocator . allocate ( outputSize );
```

Figure 4.4: The leak-inducing changes in the Snappy-Java.

Snappy-Java. Figure 4.4 shows the code excerpt which contains the leak-inducing changes in the *SnappyOutputStream* class. The *outputBuffer* is allocated from the *inputBufferAllocator*, however it is released to the *outputBufferAllocator*.

After applying our approach to Snappy-Java, it could catch the instantiation site of the *inputBufferAllocator* as a suspicious allocation site with a high leak confidence score and also assigned it the highest rank in the list of suspects.

### Solr

Issue Solr-1042<sup>12</sup> reports a memory leak in the *DataImportHandler*. If *SqlEntityProcessor* executes *DataImport* many times, the instances of *TemplateString* will be cached. This causes the memory footprint to grow steadily until it reaches an *OutOfMemory* exception. The solution for this memory leak is to use the *TEMPLATE\_STRING* as a non-static variable. This is applied by the developers to the next version of *SOLR*:

```
- private static final TemplateString TEMPLATE_STRING = new TemplateString();  
+ private final TemplateString templateString = new TemplateString();
```

Our approach could pinpoint the instantiation site of the *TEMPLATE\_STRING* variable as the root cause of this memory leak by ranking it among the top four potential root causes. The leak confidence score of the root cause was 0.77 - a small difference to the site with the highest leak confidence score (0.83).

### Nutch

Issue Nutch-925<sup>13</sup> reports a severe memory leak in the plugin repository cache used in the *PluginRepository* class of *NUTCH*. The *plugins* are stored in a *WeakHashMap* *<conf, plugins>*. Each time *plugins* are needed, a new *Class* and *ClassLoader* will be

<sup>12</sup><https://issues.apache.org/jira/browse/SOLR-1042>

<sup>13</sup><https://issues.apache.org/jira/browse/NUTCH-925>

created. Since *Class* and *ClassLoader* are stored in the permanent heap space they cannot be garbage collected. Therefore the *OutOfMemory* exception will be thrown eventually. Following is the partial changes applied by the developers to fix this issue:

```

- private static final WeakHashMap<Configuration, PluginRepository> CACHE =
-   new WeakHashMap<Configuration, PluginRepository>();
+ private static final WeakHashMap<String, PluginRepository> CACHE =
+   new WeakHashMap<String, PluginRepository>();

```

We applied our leak confidence analysis to this memory leak. Our approach could isolate the leak root cause among the top eight entries of the ranked list with a relatively high leak confidence score of 0.8.

## Thrift

Issue Thrift-1468<sup>14</sup> reports a memory leak which is captured during the running of Apache HCatalog. According to the description of this issue, if a HCatalog server runs for a long time with continuous client requests, the memory footprint of the *metastore-server* grows continuously until it reaches an *OutOfMemory* exception. The *HCatalog-server* uses Apache Thrift. There is a *WeakHashMap* which maps *TTransport* objects to their wrapped *TSaslServerTransport* instances in the class *TSaslServerTransport* of the Apache Thrift. However, in the *WeakHashMap* the value has a hard reference back to the key. Therefore the entry persists for all time, since the key can not be garbage collected. This causes an increase in the memory usage when the *HCatalog-server* is running. Following is a part of the patch that is applied by the developers to fix this issue:

```

- private static Map<TTransport, TSaslServerTransport>
-   transportMap = Collections.synchronizedMap(new WeakHashMap<TTransport,
-   TSaslServerTransport>());
+ private static Map<TTransport, WeakReference<TSaslServerTransport>>
+   transportMap = Collections.synchronizedMap(new WeakHashMap<TTransport,
+   WeakReference<TSaslServerTransport>>());

```

<sup>14</sup><https://issues.apache.org/jira/browse/THRIFT-1468>

Table 4.6: The contribution of each factor in leak confidence analysis of real cases. Section (a) shows the value of each factor for the leaky allocation site of each of the cases. Section (b) reports the rank of each leaky allocation site in the ranked list using each factor.

Issue	(a) <i>LC</i> Value				(b) Rank			
	$A(as)$	$B(as)$	$C(as)$	$LC(as)$	$A(as)$	$B(as)$	$C(as)$	$LC(as)$
hadoop-8632	0.99	0.99	0.99	0.98	15	1	107	2
hdfs-5671	0.9	0.67	0.9	0.54	552	1777	2662	668
yarn-1382	0.82	0.97	0.99	0.79	761	319	51	115
snappy-91	0.92	0.67	0.92	0.57	1	13	1	1
solr-1042	0.92	0.92	0.92	0.77	5	1	11	4
nutch-925	0.93	0.93	0.93	0.8	14	5	25	8

However, our approach was not successful in the case of Apache Thrift. After further investigations, we found that the correspondent function triggering this memory leak was not exercised by the test suites provided with the Apache Thrift. Naturally, our approach was unable to isolate the root cause.

This memory leak is highly environment-sensitive, and hence it can be only triggered by exercising a specific pattern. Although this case shows a limitation of our method, it simultaneously points possible extensions. We can use test generation techniques with optimization objective to find patterns triggering memory leaks. This is an important research direction which is also mentioned by the previous work [96, 130].

#### 4.4.3 Answer to RQ3: Analysis of Factors Contributing to LC

To understand the contribution of each factor in the equation 4.1 on the result of leak detection, we perform the leak confidence analysis by incrementally applying of each factor in the equation 4.1. Table 4.6 shows the result of this evaluation for real cases.

From the results, we can observe that all three factors can contribute to the performance of leak confidence analysis. However, the contribution of each factor to the  $LC$  value depends on the project. For example, the value of  $B(as)$  factor is directly affected by the number of unit tests which their resulting heap profiles contain the allocation site  $as$ . Therefore it is clear to see that the more unit tests are executing allocation site  $as$ , the higher value for  $B(as)$  factor will be obtained. This

value is lowest for the leaky allocation site in HDFS-5671 and Snappy-91 because only two unit tests exercised the leak path in these cases.

#### 4.4.4 Answer to RQ4: Evaluation of Runtime and Memory Efficiency

To answer RQ4, we measured the performance of our approach using POSIX-conform operating system commands. *Runtime* and *Resident Set Size* (RSS) are the metrics that we collected for each unit test.

To compute the overhead of our approach, we collected the runtime and RSS after the execution of each unit test with and without instrumentation. Then for each subject programs we aggregated the obtained results from execution of all of the unit tests corresponded to the subject program. The overhead is then computed as follows:

$$overhead = \frac{\sum_{ut \in UT} metric_{inst} - \sum_{ut \in UT} metric_{w/o inst.}}{\sum_{ut \in UT} metric_{w/o inst.}}$$

where metric can be the runtime or RSS ( $metric_{inst}$  is the value of instrumented version, and  $metric_{w/o inst.}$  without instrumentation) and  $UT$  is the set of unit tests corresponding to each application.

Figure 4.5 shows the runtime and RSS overhead of our approach on the subject programs. The result shows that our approach imposes a moderate overhead on both runtime and RSS which makes it applicable for the development phase.

The overhead runtime of our approach is imposed by the instrumentation used for collecting the heap profiles. It causes delays in both instantiation and deletion of allocation objects which increases in the overall execution time of the unit tests. As shown in Figure 4.5, the execution time of the unit tests with instrumentation is from 0.55 to 3.75 times more than the runtime of the unit tests without instrumentation. One solution to decrease the runtime overhead is the selective instrumentation. For example, we can instrument only the part of the code which is only relevant to the recent code changes.

Resident set size (RSS) is also measured for each of the subject programs. Figure 4.5 shows that the aggregate RSS for each subject programs with

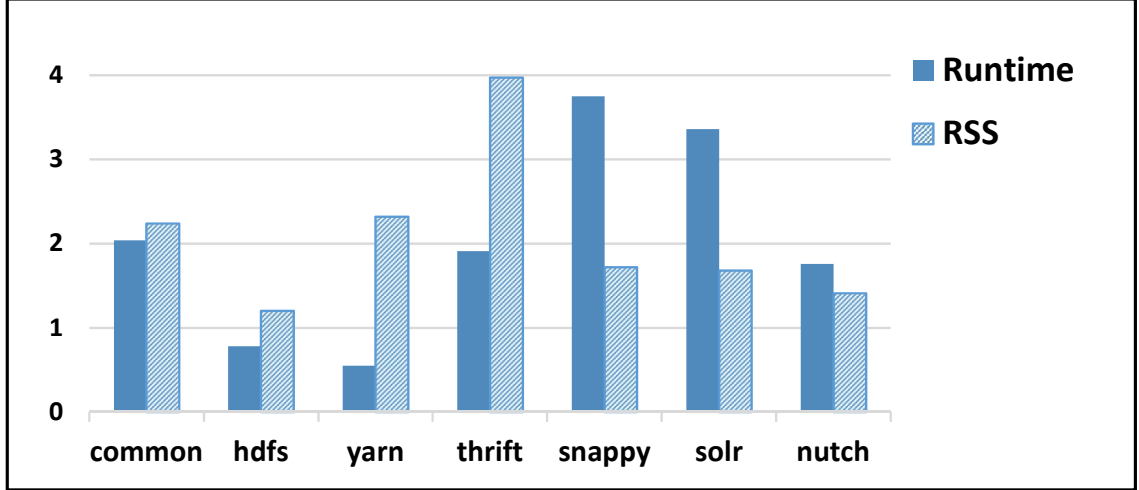


Figure 4.5: Runtime and RSS overhead of subject programs. Overhead of 1 ( $y$ -axis) means that the instrumented version has twice the runtime or RSS of the non-instrumented version.

instrumentation is from 0.9 to 3.97 times more than aggregate RSS of that program without instrumentation.

## 4.5 Discussion

### 4.5.1 What is the Distribution of the Leak Confidence Value for Various Software Projects?

In this section, we analyze the distribution of the leak confidence value for various software projects. In a scenario where an allocation site with a highest leak confidence score is used to alert about the potential presence of leaks automatically, knowledge of this distribution is essential to find a value of a threshold  $LC_{th}$  which balances the rate of false positives and false negatives. We assume here a simple alerting mechanism where an alert is triggered if the highest leak confidence score is above a threshold  $LC_{th}$ . A low threshold value can introduce many false positives, while high threshold values might lead to false negatives (i.e., missing leaky allocation sites).

For this, we plotted the probability density function of leak confidence value for the allocation sites reported in the ranked list of suspects for each application. Figure 4.6 shows the results of this evaluation for real memory leaks. We can see that for most

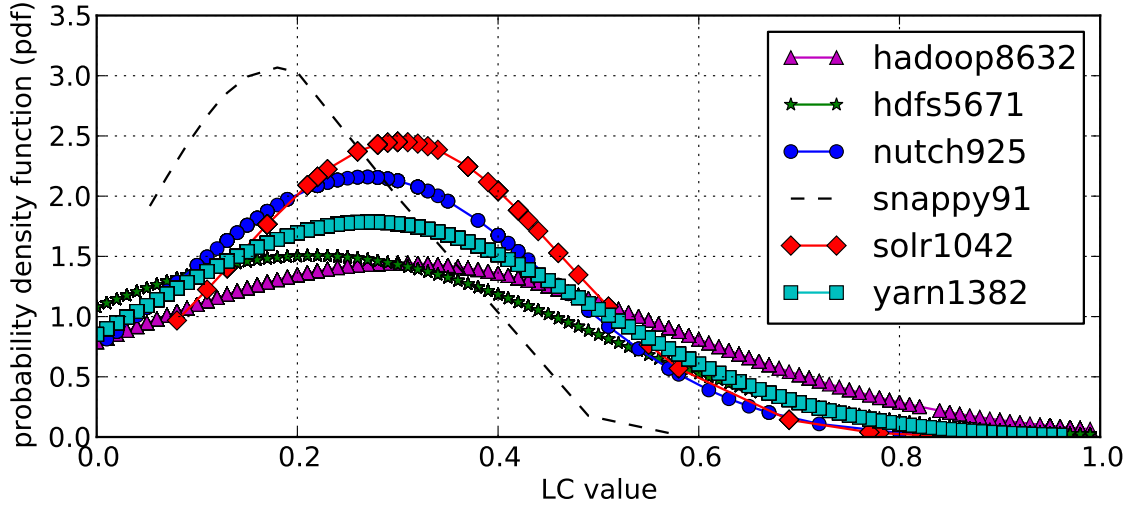


Figure 4.6: Distribution of leak confidence score  $LC$  for the allocation sites reported in the ranked list for real leaks.

projects except snappy the distribution is similar: it peaks around  $LC$  value of 0.3, and it is significantly tailed towards the larger  $LC$  values (between 0.8 and 1.0). For snappy, the peak is at 0.18 and largest  $LC$  values are in the range 0.55 - 0.6.

In comparison with Table 4.5, we conclude that such value of the threshold  $LC_{th}$  in the interval 0.7 - 0.8 is reasonable for most but not all projects. Therefore a “reasonably useful” automated detection of the presence of leaks based on a simple threshold is possible, but only after a careful investigation of specifics of the targeted software artifact.

#### 4.5.2 Does Our Approach Help Developers to Detect Memory Leaks?

Leak detection (i.e., alerting about the existence of any potential new leaks) can be performed in two ways. The first option is manual inspection of the top-ranked allocation sites in the ranked list. If the leak confidence value of the top entries is substantially higher than previously observed values for this application, or if previously unknown allocation sites surface to the top, a manual alert can be triggered.

#### 4. Automated Memory Leak Diagnosis via Version Comparison

The second option is to raise an alert automatically if the highest  $LC$  value in the list is above a threshold, i.e.,  $LC(as) > LC_{th}$ . The optimal value of such a threshold needs to be adjusted for each application separately. After the investigation on real-world projects, we found that the threshold for  $LC$  value is project-based. It means that we can not set the same threshold value for any arbitrary project. Section (b) of Table 4.5 shows the actual leak confidence values of the leak-inducing allocation sites (column  $LC$ ). For comparison, we also included  $LC_{max}$  as the largest leak confidence value among all sites in the ranked list. The values reported for  $LC_{max}$  clearly show that the top values of the leak confidence scores vary between applications, ranging between 0.54 (Snappy-91) and 0.99 (HADOOP-8632, HDFS-5671). This makes it indeed necessary to use application-specific threshold values for automated (threshold-based) leak detection.

As shown in the Section 4.5.1, it is possible to use our approach for memory leak detection. However, much more sophisticated methods such as adaptive thresholds or a combination of a threshold and “novelty” of top-ranked sites are needed to increase the accuracy of the leak detection. The investigation of such schemata is beyond the scope of this work.

### 4.5.3 Can Our Approach Find the Root Cause of the Memory Leaks?

Leak isolation, finding the root cause of memory leak is performed analogously to automated debugging: a developer is given a ranked list of suspects and investigates the code and behavior of the top-ranked allocation sites in this list.

As discussed in Section 4.4, our method is acceptably accurate by placing the leak-inducing allocation sites close to the top of the ranking list. After further investigation of the real-world cases, we found out that in most of the cases, the developers mainly modified the leak-inducing allocation sites (pinpointed by our approach) to fix memory leaks. This is the case for issues HADOOP-8632, YARN-1382, HDFS-5671, Solr-1042, and Nutch-925. In Snappy-91, the developers fix the memory leak issue with changing source code lines different than the reported top-ranked suspicious allocation sites in the ranked list. However, after



digging into the source code, we realized that the reported leaky allocation site is the instantiation site of the modified line. Therefore our approach indirectly pinpointed the root cause of the memory leak. In such cases, a possible solution for finding the primary root cause is using common IDEs or some static analysis approaches such as forward slicing (with instantiation site as seed statement).

## 4.6 Threats to Validity

**Threats to external validity** arise when the results of the experiments cannot be generalized for any arbitrary program. We evaluated our approach on the limited number of applications. Therefore we can not claim that our approach can isolate memory leaks in all types of real-world applications. However, we are confident that our approach can be applied to a variety of Java applications (and C/C++ applications with suitable code instrumentation). This can help developers in the root cause analysis of memory leaks and also narrowing down the list of suspicious allocation sites.

**Threats to construct validity** arise when our approach is unable to pinpoint the leaky allocation site because of lack of the tests which trigger memory leak pattern. It means that the accuracy of our approach is influenced by the ability of the tests to trigger the leak-inducing defects. We can avoid this problem by having test suites with better code coverage which is a common goal in the quality assurance of current software projects.

**Threats to internal validity** arise when our approach instrument the source code of the application in question. Delay in the instantiation and the destruction of the allocated objects due to the code instrumentation might affect the run time of the testing process. However, the overall overhead of our approach is moderate and is comparable to the existing leak detection approaches.

## 4.7 Chapter Summary

In this chapter, we proposed a regression-based leak detection technique. We leveraged dynamic analysis to generate the profile of the allocations and

#### 4. Automated Memory Leak Diagnosis via Version Comparison

deallocations for each allocation site in the codebase of the application during runtime. Then, we compared the resulted profiles between two previous (non-leaky) and current (leaky) versions of the application to find the suspicious allocation sites. Then, using a newly introduced confidence score, we rank the suspicious allocation sites. The top-ranked allocation sites are highly likely potential root causes of the memory leaks.

From the results of the experiments, we found out that our approach is generally useful in the detection and isolation of memory leaks. By analyzing the result obtained from the first experiment (i.e., experiments on synthetic leaks), we can conclude that our approach can diagnose memory leaks if the (unit) tests exercise the execution pattern triggering the leak. Therefore, we can assess for RQ1 that *our approach is precise in finding the synthetic leaky allocation sites.*

In terms of effectiveness on the real-world cases, we examined our approach on several mid to large real-world applications (also to verify the scalability of our detection approach). From six out of seven cases, our approach included the leaky allocation site in the list of suspects. In four cases, the root cause of memory leaks was ranked in the top 10 of the ranking report. As such, for RQ2, we can assess that *our approach applies to large, real programs and it can diagnose defects if the tests code covers the leak-inducing allocation site.*

We also evaluated the contribution of the factors used in the leak confidence metric. As such, for RQ3, we can assess that *each of the three factors contribute to the leak confidence metric. However, the effect of each factor on the leak confidence value depends on the application.*

Based on the measured data we can assess for RQ4 that *our approach imposes acceptable overhead in terms of runtime and memory consumption. This makes our approach feasible for use in the testing process.*

Our proposed approach also differs from previous related work [65]. Compared to this work, we provided a more robust and accurate result using a leak confidence analysis. In this chapter, we used a refined metric, mainly the (absolute) numbers of allocated and deallocated objects measured for many different (unit) tests to find the suspicious allocation sites statistically while in work [65] the authors only used a single unit or integration test for finding the suspicious allocation sites.

## Chapter 5

# An Empirical Study on Leak-inducing Defects and Their Repairs

Despite many software engineering efforts and programming language support, resource and memory leaks remain a troublesome type of issues, even in managed languages such as Java. Understanding the properties of leak-inducing defects, how the leaks manifest and how they are repaired is an essential prerequisite for designing better approaches for avoidance, diagnosis, and repair of leak-related bugs.

We conduct a detailed empirical study on 491 issues from 15 mature and large Java projects. We propose taxonomies for the leak types, for the defects causing them, and for the repair actions. We investigate, under several aspects, the distributions within each taxonomy and the relations between them. Based on our findings we draw a variety of implications how developers can avoid, detect, isolate and repair leak-related bugs.

### 5.1 Introduction

Leaks are unreleased system resources or memory objects which are no longer used by an application. In memory-managed languages such as Java, C#, or Go, a garbage collector handles memory management. The garbage collector uses object reachability to estimate object liveness. It disposes of any heap objects which are no longer reachable by a chain of references from the root objects. However, if an unused

object is still reachable from other live objects, the garbage collector cannot reclaim the space. Aside from memory, finite *system resources* such as file handles, threads, or database connections require explicit management specified in the code. It is the responsibility of the programmer to dispose of the acquired resource after using it; otherwise, a resource leak is likely.

Leak-related bugs are severe [115] and can finally result in performance degradation and program crash. Hence, they should be resolved at an early stage of development. However, due to their non-functional characteristics, leaks are likely to escape traditional testing processes and become first visible in a production environment. The root cause of a memory leak can differ from the allocation which exhausts the memory [59]. Some leaks can only be triggered if abnormal behavior occurs such as an exception or a race condition. These factors make leak diagnosis laborious and error-prone.

Defects induced by memory and resource leaks are among the important problems for both researchers and practitioners. Microsoft engineers consider leak detection and localization as one of the top ten most significant challenges for software developers [75]. This problem is addressed by various researchers, tools, and programming languages. Many previous works targeted memory and resource leak diagnosis by leveraging static and dynamic analysis. Static analysis is used for leak detection via finding unclosed resources on different execution paths [26, 34, 106, 116, 121, 133]. The main challenge of static analysis is the lack of scalability and high rate of false positives. To mitigate this issue, researchers apply dynamic analysis techniques for leak diagnosis [14, 37, 50, 59, 88, 94, 131].

Programming languages provide support for programmers to prevent occurrences of leak-inducing defects. For instance, Java 7 introduces a new language construct, called `try-with-resources`<sup>1</sup> to dispose of the objects that implement the *autoclosable* interface. Various open-source or proprietary tools (e.g., FindBugs<sup>2</sup>, Infer<sup>3</sup>) also aim to help programmers to find the potential leaks in the software codebase. For example,

---

<sup>1</sup><https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>

<sup>2</sup><http://findbugs.sourceforge.net>

<sup>3</sup><http://www.fbinfer.com>

FindBugs provides some rules<sup>4</sup> to warn programmers about potential file descriptor leaks.

Despite the above-mentioned academic works, language enhancements, and tool supports, several challenges are still open. The impact of these efforts depends on whether they target common or rare issue types, whether they can handle severe cases, and whether their assumptions are realistic enough to be applicable in practice. Programming language enhancements such as `try-with-resources` or tool support such as FindBugs help to find only the resource leaks and no memory leaks. Many of the academic works are motivated by anecdotal evidence or by empirical data collected only from small sets of defects. For example, [131] propose a method for detecting memory leaks caused by obsolete references from within object containers but provide only limited evidence that this is a frequent cause of leak-related bugs in real-world applications. As another example, Leakbot [88] introduces multiple sophisticated object filtering methods based on observations derived from only five large Java commercial applications.

A systematic empirical study of a large sample of leak-related defects from real-world applications can help both researchers and practitioners to have a better understanding of the current challenges on leak diagnosis. We believe such a study can be beneficial in the following directions:

**Benefit 1.** A representative study can characterize the current approaches for leak diagnosis used in practice. This can guide researchers to find limitations of leak detection approaches and motivate further improvements. The results would provide a comprehensive basis for the design and evaluation of new solutions.

**Benefit 2.** It helps programmers to avoid mistakes made by other programmers and shows some of the best practices for leak diagnosis.

**Benefit 3.** It can be used as a verification for the assumptions used in previous work. For example, it is interesting to verify whether there is a large number of leaks caused by collection mismanagement in real-world applications. The definite answer to this could confirm the assumption of [131] on memory leak detection.

To the best of our knowledge, the research body of empirical studies on resource and memory leak-related defects is relatively thin in comparison with the vast body

---

<sup>4</sup><http://findbugs.sourceforge.net/bugDescriptions.html>

## 5. An Empirical Study on Leak-inducing Defects and Their Repairs

of studies about other bug types (e.g., semantic or performance bugs). The existing studies [78, 115] provide only limited information about characteristics of detection types, root causes, and repair actions of leak defects. To fill this gap, we conduct a detailed empirical study on 491 real-world memory, and resource leak defects gathered from 15 large, open-source Java applications [42, 43].

We manually study the collected issues and their properties: leak types, detection types, common root causes, repair actions, and complexity of fix patches. Based on our findings, we draw several implications on how to improve avoidance, detection, localization, and repair of leak defects. In particular, this study tries to answer the following research questions:

- . **RQ1.** What is distribution of leak types in studied projects?
- . **RQ2.** How are leak-related defects detected?
- . **RQ3.** To what extent are the leak-inducing defects localized?
- . **RQ4.** What are the most common root causes?
- . **RQ5.** What are the characteristics of the repair patches?
- . **RQ6.** How complex are repairs of the leak-inducing defects?

This work provides the following contributions:

**Characterization study.** We conduct an empirical study on 491 bugs from 15 mature, large Java applications. To the best of our knowledge, this is the first work which studies characteristics of leak-related bugs from real-world applications comprehensively while using a broad set of issues from diverse open-source applications.

**Taxonomies.** We propose taxonomies for leak types (Section 5.4.1), detection types and methods (Section 5.4.2), root causes (Section 5.4.4), and repair actions (Section 5.4.5).

**Analysis.** We investigate the distributions of leaks across the categories within each taxonomy and the relation between the taxonomies. Our findings show that source code analysis and resource monitoring are the main techniques to detect leaks. Our analysis using a state-of-the-art resource leak detection tool (i.e., Infer) highlights that

the static analysis tools require further improvement to detect different leak types in practice. We find that 76% of the leaks are triggered during the error-free execution paths. We identify 13 recurring code transformations in the repair patches. We also show that developers resolved the studied issues in about six days on the median.

**Implications.** We use our findings to draw a variety of implications on the leak prevention and diagnosis for both researchers and practitioners (Section 5.5).

**Replicability.** To make our study replicable and reusable for the community, we make the dataset and the results available online<sup>5</sup>.

## 5.2 Background

### 5.2.1 Issue Report

Modern projects often use an Issue Tracking System (ITS) to collect the issues reported by users, developers, or software quality teams. An issue typically corresponds to a bug report or a feature request. Bugzilla<sup>6</sup>, JIRA<sup>7</sup>, and GitHub issue tracker<sup>8</sup> are examples of ITS systems. Each issue report in the bug tracker is identified with a unique identifier. For example, in JIRA, this is a combination of the project name and a number (e.g., SOLR-1042). In GitHub, an identifier is a number with a preceding hashtag (e.g., issue #1865 in RxJava project). An issue report in Jira contains a variety of information such as title, description, comments, and links to the related fix patches. It also contains metadata information such as type, status, priority, resolution, and associated timestamps (e.g., created or resolved timestamps). Figure 5.1 shows a snippet of an issue report from Jira. All the information provided in issue reports makes the issue tracker a rich environment to get more insights on bugs and their corresponding repairs.


---

<sup>5</sup>[https://github.com/heiqs/leak\\_study](https://github.com/heiqs/leak_study)

<sup>6</sup><https://www.bugzilla.org/>

<sup>7</sup><https://issues.apache.org/jira/projects/>



<sup>8</sup><https://github.com/>


Derby / DERBY-1142

## Metadata calls leak memory

---

### Details

Type:	 Bug	Status:	<span>CLOSED</span>
Priority:	 Minor	Resolution:	Fixed
Affects Version/s:	10.1.2.1, 10.2.1.6	Fix Version/s:	10.2.1.6
Component/s:	JDBC		
Labels:	None		

---

### Description



When calling a DatabaseMetaData method that returns a ResultSet, memory is leaked. A loop like this (using the embedded driver)

```
while (true)
{ ResultSet rs = dmd.getSchemas(); rs.close(); }
```

will eventually cause an OutOfMemoryError.

---

### Attachments


 <a href="#">1142_close_single_use_activations_draft.txt</a>	2 kB	10/Jul/06 21:17
 <a href="#">metadataloop.java</a>	0.6 kB	23/Mar/06 03:08

---

### Activity

All
**Comments**
Work Log
History
Activity
Transitions

---

 [Knut Anders Hatlen](#) added a comment - 23/Mar/06 03:08

Attached repro. With Derby 10.1.2.1 and Sun JVM 1.4.2, OutOfMemoryError was thrown after about 80000 calls to DatabaseMetaData.getCatalogs().

Figure 5.1: An issue report from JIRA.

## 5.3 Empirical Study Design

In this section, we describe the design of our empirical study. Figure 5.2 gives an overview of our methodology. In the remainder of this section, we illustrate the research questions, studied applications, and data collection process.

### 5.3.1 Studied Projects

We perform a study on 15 open-source Java projects hosted in two major repositories, Apache and GitHub. We investigate the leak-related issues from a wide



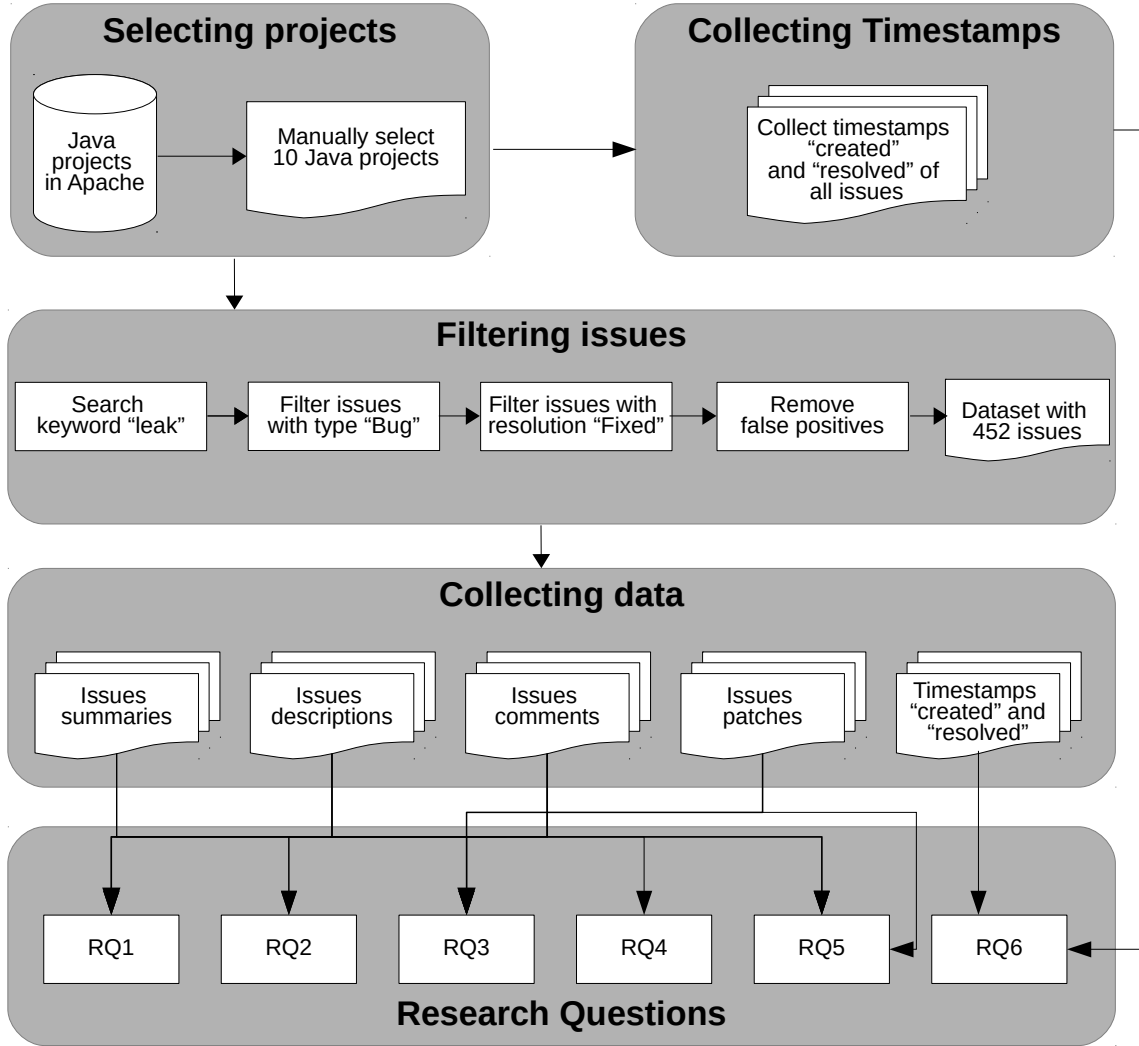


Figure 5.2: Overview of the empirical study design.

variety of software categories to ensure the diversity of the studied projects. Table 5.1 lists the studied projects. AMQ<sup>9</sup> is an open-source message broker with the support of cross language clients and protocols. CASSANDRA<sup>10</sup> is a distributed database targeting high scalability and availability. CXF<sup>11</sup> is an open source framework for developing services using frontend programming APIs. DERBY<sup>12</sup> is an open-source relational database. HADOOP<sup>13</sup> is a distributed computing

<sup>9</sup><http://activemq.apache.org>

<sup>10</sup><http://cassandra.apache.org>

<sup>11</sup><http://cxf.apache.org>

<sup>12</sup><http://db.apache.org/derby>

<sup>13</sup><http://hadoop.apache.org>

Table 5.1: Overview of studied projects. Column “Since” lists the year of the first commit and column “#Committers” presents the number of committers in each projects based on Apache Committee Information. The kLOC of each project shows the number of Java code lines retrieved by OpenHub.

Project	Category	Since	#Committers	#kLOC
AMQ	Distributed messaging	2004	58	1158
CASSANDRA	Distributed database	2009	45	313
CXF	Web service	2007	38	674
DERBY	Relational database	2004	44	689
HADOOP	Distributed computing	2006	163	1260
HBASE	Distributed database	2007	57	1115
HIVE	Data warehouse	2009	63	1074
HTTPCOMP.	Network client/server	2004	18	115
LUCENE	Search framework	2004	67	557
SOLR	Search framework	2008	67	416
Realm Java	Mobile database	2012	14	116
Spring Boot	Application framework	2012	180	311
Logstash	Data Processing	2009	43	74.6
RxJava	Reactive programming	2013	65	279
Selenium	Browser driver	2006	115	703

platform including four main components: Ha Common, HDFS, MapReduce, and YARN. HBASE<sup>14</sup> is a distributed, scalable and big data store. HIVE<sup>15</sup> is an SQL-enabled data warehouse for large datasets. HTTPCOMPONENT<sup>16</sup> with its two components core and client is a toolset for working with the HTTP protocol. LUCENE<sup>17</sup> is a high performance, cross-platform text search engine. SOLR<sup>18</sup> is an open-source full-text enterprise search server based on LUCENE. Realm Java<sup>19</sup> is the Java implementation of the Realm project which is a mobile database. Spring Boot<sup>20</sup> is a framework for creating stand-alone Spring based applications. Logstash<sup>21</sup> is a server-side data processing pipeline which transports and processes the logs, evenets, other types of data. RxJava<sup>22</sup> is a Java implementation of the

<sup>14</sup><http://hbase.apache.org>

<sup>15</sup><http://hive.apache.org>

<sup>16</sup><http://hc.apache.org>

<sup>17</sup><http://lucene.apache.org/core>

<sup>18</sup><http://lucene.apache.org/solr>

<sup>19</sup><https://github.com/realm/realm-java>

<sup>20</sup><https://github.com/spring-projects/spring-boot>

<sup>21</sup><https://github.com/elastic/logstash>

<sup>22</sup><https://github.com/ReactiveX/RxJava>

Reactive Extensions which is an API for asynchronous programming. Selenium<sup>23</sup> provides tools and libraries for the web browsing automation.

We study these projects for two reasons. First, they are large-scale and open-source projects with a mature codebase with years of development. We believe that by using such a well-established and well-developed applications, we can get results representative for mature Java projects. Column #kLOC in Table 5.1 shows the size of the Java source code of the studied projects ranging between 74 to over 1200 kLOC. For the Github projects, the total number of stars is more than 100k.

Second, their issues are reported and tracked in a bug tracking system. Similar to other bug trackers (e.g., Bugzilla), reports in JIRA or GitHub bug tracker are well-described and provide sufficient information to answer the research questions investigated in this study.

### 5.3.2 Research Questions

The following research questions guide our study:

**RQ1. What is distribution of leak types in studied projects?** In Section 5.4.1, we analyze the dominant leak types in each project. We use this analysis in the next research questions to distinguish the properties of different leak types.

**RQ2. How are leak-related defects detected?** Understanding different detection types can help leak detection approaches to improve detection accuracy. In Section 5.4.2, we investigate how developers or users report the leak-inducing defects and how the leaks manifest at runtime. We analyze different detection and manifestation types and study their relation to the leak types.

**RQ3. To what extent are the leak-inducing defects localized?** Bug localization is the first step in bug diagnosis. The extent of the bug can highly affect the number of files that need to be fixed to repair the bug. In this question, we analyze the locality of leak-inducing defects (Section 5.4.3).

**RQ4. What are the most common root causes?** Section 5.4.4 describes the common root causes of leak defects. We investigate the prevalence of each root cause and their relation to the leak types.

---

<sup>23</sup><https://github.com/SeleniumHQ/selenium>

**RQ5. What are the characteristics of the repair patches?** In Section 5.4.5, we identify the repair actions applied by the developers to repair the leak-related defects and investigate the frequency of each considering different leak types. We also search to find recurring code transformations in the repair patches. We identify 13 common repair patterns from our dataset. In this question, we investigate whether the automated program repair techniques (i.e., the process of providing the repair patches automatically) such as template-driven patch generation are applicable for fixing the leak-related defects.

**RQ6. How complex are repairs of the leak-inducing defects?** In Section 5.4.6, we measure the code churn, change entropy, and diagnosis time to assess the complexity of the changes needed to repair the leak-inducing defects. This analysis provides insights about the difficulty of repairing the leak-related defects and shows which type of leaks can be repaired with less effort in terms of time and amount of code changes.

### 5.3.3 Data Extraction

For the Apache projects, we collected the leak-related issues from the bug tracker reported until June 2016. For GitHub projects, we collected the issues reported until January 2019.

To build a suitable dataset for our study, we apply a four-step filtering methodology: (1) keyword search, (2) issue type filtering, (3) resolution filtering, and (4) manual investigation. This four-step filtering method yields a dataset with 491 leak-related issues, each representing a unique leak bug report (i.e., none are duplicates of another). We make the dataset available online<sup>24</sup>.

**Keyword search.** We use a simple heuristic and select issues that contain the keyword “leak” in the issue title or issue description. The keyword search is a popular method used by previous empirical studies [57, 92, 147] to filter the issues of interest from the others. It is worth mentioning that we investigated other leak-related keywords (unreleased, out-of-memory, OOM, closed, and others). However, these keywords yield a dataset with a high number of false positives. For example, the keyword “unreleased” is used in the title of the issue report

---

<sup>24</sup>[https://github.com/heiqs/leak\\_study](https://github.com/heiqs/leak_study)

CXF-7776<sup>25</sup>: “Download page should not link to the unreleased code.” It is evident that this issue has no relation to this study. Pruning of such issues is time-consuming and requires a considerable amount of manual effort.

On the other hand, it is possible that we skip some leak-related issues due to our simple keyword search. For example, YARN-5257<sup>26</sup> refers to some unreleased resources which are fixed. Although this is a leak-related issue, the term leak is not mentioned in the issue title or description.

Despite the simplicity of keyword search, this heuristic proved to be highly precise due to the high quality of issue reports and related data in the studied projects. [128] highlight that even simple heuristics can yield the same precision and recall as more sophisticated search techniques when applied to a well-maintained bug tracker. Using the keyword search, we identify 1255 leak-related issues. Column “#Issues” in Table 5.2 shows the number of filtered issues for each project.

**Issue type filtering.** Each issue in the bug tracker can be classified as “Bug”, “Task”, “Test”, and so on. As we are only interested in leak-related bugs, we first filter issues with type “Bug”. Among the 1255 issues filtered by keyword search, there are 838 issues labeled as a bug (column “#Bugs” in Table 5.2).

**Issue resolution filtering.** To analyze how developers repair a leak defect we need to restrict our analysis to fixed bugs. For this, we filter issues with the resolution label “Fixed” for Apache projects and “Closed” for GitHub projects. This reduces the dataset to 591 issues (column “#Fixed” in Table 5.2).

**Manual investigation.** In the final step, we remove the false positives from our dataset. We manually filter out the following issues:

- Non-leak-related bugs retrieved by our keyword search heuristic. For instance, in issue CXF-3390<sup>27</sup>, the term *leak* is used in “information leak” which is not related to this study.
- Wrongly reported leaks. These issues should be tagged as “Invalid”, but are closed in the bug tracker without correct labeling.

---

<sup>25</sup><https://issues.apache.org/jira/browse/CXF-7776>

<sup>26</sup><https://issues.apache.org/jira/browse/YARN-5257>

<sup>27</sup><https://issues.apache.org/jira/browse/CXF-3390>

Table 5.2: Studied projects with statistics on number of issues (explained in Section 5.3.3). Columns “#MLeak”, “#RLeak”, and “Total” show the numbers of memory and resource leak issues per application, and their totals, respectively.

Project	#Issues	#Bugs	#Fixed	#MLeak	#RLeak	Total
AMQ	123	116	88	54	26	80
CASSANDRA	77	65	45	19	16	35
CXF	62	61	44	29	8	37
DERBY	50	36	23	12	4	16
HADOOP	236	201	132	43	76	119
HBASE	92	65	44	11	29	40
HIVE	78	69	47	19	25	44
HTTPCOMP.	31	28	24	8	12	20
LUCENE	77	65	42	13	21	34
SOLR	74	60	33	11	16	27
Realm Java	76	15	15	4	2	6
Spring Boot	94	17	16	2	10	12
Logstash	67	25	23	8	4	12
RxJava	100	14	14	5	3	8
Selenium	18	1	1	0	1	1
<b>Total</b>	<b>1255</b>	<b>838</b>	<b>591</b>	<b>238</b>	<b>253</b>	<b>491</b>

### 5.3.4 Tagging Leak-Related Defects

To analyze the properties of the leak-related defects, we need to classify the issues for each dimension of interest (i.e., leak type, detection type, detection method, defect type, and repair type). However, we only have qualitative information such as issue description, developers discussions, and repair patches. There is no label provided in the bug tracker for classification of the attributes that we are interested in reported leaks. To derive properties for the bugs in our dataset, we need to quantify the qualitative information. For this purpose, we perform an iterative process similar to *Open Coding* [103, 104]. In our study, the input of the coding process for each issue is issue summary, issue description, developers discussions, and repair patches.

Work [43] explains the process of the tagging in details. We first classify a sample set of issues to determine the possible categories for each dimension. After identifying the primary types for each category, we continue to label other issues based on the preliminary categories.

The tagging process is iterative. Each time a new type is identified, the coders (the first three authors in work [43]) verify it in a decision-making meeting. If a

Table 5.3: Cohen’s kappa measurement.

Dimension	Cohen’s Kappa
Leak Type (RQ1)	0.86
Detection Type (RQ2)	0.83
Detection Method (RQ3)	0.70
Defect Type (RQ4)	0.69
Repair Type (RQ5)	0.57

majority of the coders agree on the new type, they go through all the previously tagged issues and check if the issues should be tagged with the new type. This also minimizes the threat of human error during the labeling process. To further reduce the error probability and in case of difficulty in classifying of the issues, all the coders check and discuss the complex issues to find the appropriate categories. The conflicts were resolved by discussing and coming to an agreement.

To validate the manual labeling process, we apply the following procedure. The first two coders perform a classification of a statistically representative sample of the dataset with a confidence level of 95% and a confidence interval of 10%. This results in a sample set of 80 out of 491 issues. We calculate the inter-rater agreement with Cohen’s kappa metric [9, 30]. Table 5.3 shows the result of our analysis. The lowest Cohen’s kappa value is for the repair type, although it shows a moderate agreement between the two coders. The reason for disagreements is that the categories in this attribute are not mutually exclusive. Therefore, there is a probability that each coder has a different interpretation of the same issue. After rating, the two raters discussed their disagreements to reach consensus.

### 5.3.5 Uniqueness of Categories

During tagging task, we encounter the issues with the possibility of assigning them to multiple categories. For example, in Hadoop-6833<sup>28</sup>, a leak is reported due to the forgotten call to the *remove* method of a collection. The developers repaired the bug by adding the remove call in the exception path:

```
--- src/java/org/apache/hadoop/ipc/Client.java
```

<sup>28</sup><https://issues.apache.org/jira/browse/HADOOP-6833>

```

+++ src/java/org/apache/hadoop/ipc/Client.java
@@ -697,6 +697,7 @@ public class Client {
        } else if (state == Status.ERROR.state) {
            call .setException(new RemoteException(WritableUtils.readString(in),
                                                    WritableUtils . readString(in)));
+
            calls .remove(id);
        } else if (state == Status.FATAL.state) {
            // Close the connection
            markClosed(new RemoteException(WritableUtils.readString(in),

```

One could label this issue as *collection mismanagement*. However, if the exception is thrown no leak is triggered. Therefore, we use the underlying cause as the main root-cause category (here *bad exception handling*). For the repair action, we assign a bug to the category used by the developer to repair the defect. In this example, we label the repair action as *remove element*.

## 5.4 Empirical Study Results

In this section, we answer the research questions. For each research question, we describe the motivation behind the question, the approach used in answering the research question, and the findings derived from the analysis.

### 5.4.1 RQ1: What Is Distribution of Leak Types in Studied Projects?

**Motivation.** In this research question, we want to find the primary leaked resource for each issue. The leak type classification will be used in further research questions to determine the existence of different patterns for different leak types. We also use this investigation to assess the leak diversity on the studied projects. Understanding the dominant leak type in each project can also help developers to focus on this leak type for the current development phase.

**Approach.** For most of the studied issues, the reporters or developers explicitly mentioned the leak type. For such cases, we assign the leak type as reported. In case



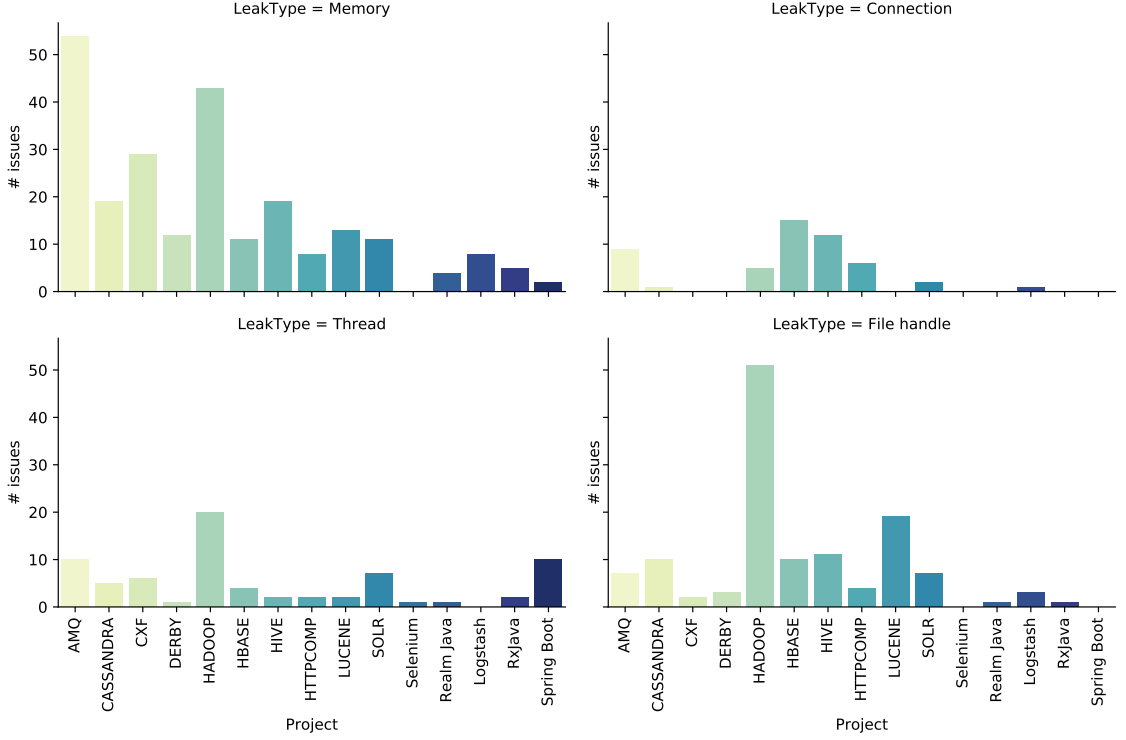


Figure 5.3: Frequency of the leak types per project.

of no explicit mention of the leak type, we manually analyze the title, description, and developers discussions to assign the correct leak type.

***Taxonomy of leak types.*** Our analysis yields a taxonomy of leak types with the following four categories:

***Memory.*** We group in this category all issues reported due to unreleased references to Java objects, such as mismanagement of collections or circular references.

***File handle.*** We group in this category leaks related to file descriptors. These issues are related to the mismanagement of Java file handlers such as I/O streams.

***Connection.*** We group in this category leaks triggered by non-closed network or database connections.

***Thread.*** We group in this category leaks caused by unclosed, yet unused threads. A thread leak occurs when a no-longer-needed thread is unnecessarily kept alive. This thread then leaks its internal resources, which cannot be released by the JVM. This is a particular case of a resource leak which always leaks memory as a consequence,

as each thread carries several local variables that can not be disposed of by the GC, while the thread is alive.

**Results.** Figure 5.3 shows the distribution of the leak types for each project. We use this data to find the dominant leak types for each project and the project categories.

**Finding 1.** The three leak types corresponding to the resource leaks (i.e., file handle, connection, and thread) is the most common leak types in six out of the ten projects. Resource leaks (with 253 issues) are slightly more reported than memory leaks (with 238 issues).

**Finding 2.** Each project shows a distinct distribution of the leak types. LUCENE and HADOOP have a higher frequency of the *file handle* leak type with this leak type corresponding to 55.9% and 42.9% of the issues, respectively. Projects AMQ (67.5%), CASSANDRA (54.3%), CXF (78.4%), and DERBY (75.5%) contain more memory leak issues. Connection leaks are more frequently reported in HBASE (37.5%), HTTPCOMP (30%), and Hive (27.3%). 10 out of 12 issues in Spring Boot are of type thread leak. This analysis shows the diversity of the leak types in the studied projects. Even projects within the same category show different distributions of the leak types.

**Summary.** Resource leaks (253 out of 491 issues) are slightly more often reported than memory leaks (238 issues). Leak type distribution is different across the projects.

#### 5.4.2 RQ2: How Are Leak-Related Defects Detected?

**Motivation.** Each issue report provides information about leak symptoms, environmental setup, and methods used to detect a leak. Understanding how leaks are detected can provide valuable insights on leak diagnosis. It also shows in which direction the researchers and tool builders should help programmers in leak detection. In this question, we want to find whether the leaks are detected during runtime and whether the static analysis is used for leak detection.

**Approach.** To find detection type for each issue, we use three data sources: issue title, issue description, and developers discussions. Using this data, we analyze the

frequency of the detection types, distribution of detection methods, and their relation to different leak types.

***Taxonomy of leak detection.*** Leak-inducing defects can be discovered with and without runtime failures or performance degradation. They can be detected via manual analysis of the source code, an unexpected runtime failure (in particular, an out-of-memory error), or abnormal usage of resources. We classify detection types into two major categories: source code-based detection and runtime detection. In the following, we explain these two detection types in more detail.

*Source code-based detection.* In this category, we classify issues such that the leak detection is performed before execution of the program and there is no reported runtime information in the issue report, nor reports on leak-related failures. We observe that issue reporters describe these issues with phrases such as “can potentially cause a leak” or “can lead to a leak”. The main techniques to detect leaks before the runtime are *manual code inspection* and *static analysis tools*.

Manual inspection of the source code (or code review) is a process in which developers inspect a set of program elements (e.g., methods, classes) in order to improve the quality of software [54, 83, 109]. It is one of the most common static detection methods used by developers in practice. This detection type requires the knowledge of how a leak can be introduced as well as an understanding of the application’s behavior. For instance, in AMQ-5745<sup>29</sup>, manual inspection revealed cases where bad exception handling could yield resource leaks on the AMQ codebase.

Static analysis tools can be used to identify potential leak defects during the development process without the need for runtime information. Such tools use the source-code or byte-code to reason about programs. There are many free and proprietary static analyzers which can detect specific leak types (e.g., FindBugs, Infer). For example, FindBugs includes two rules to detect resource leaks related to unclosed I/O streams.

*Runtime analysis.* Some leak-related failures are observed and reported when a user/developer encounters a performance degradation in a production environment, an out-of-memory error is raised, or a test is failed. Issue reporters often use phrases

---

<sup>29</sup><https://issues.apache.org/jira/browse/AMQ-5745>

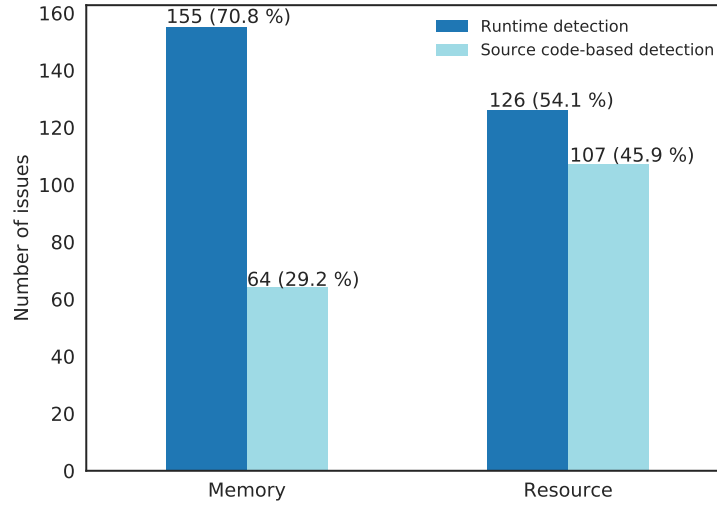


Figure 5.4: Frequency of the detection types per leak type.

such as “consistently observing memory growth” or “meet memory leak in a production environment”. In these issues, the bug reporter often provides additional material such as heap profile, memory dump, a log file, or a stack trace. This additional data can help developers on localizing the root cause of the leak defect more efficiently. Leaks usually manifest in the runtime with a symptom. From our observation, we identify three symptoms reported in the issue reports: failing tests, out-of-memory errors, and warning messages.

The output of a failing test case may expose a leak. The test can be a system test, a unit test or any other application-provided test. For example, in LUCENE-3251<sup>30</sup>, a failing unit test exposed a file-handle leak. The user provided the stack trace of the failing test in the issue report:

```

Testsuite : org.apache.lucene.index.TestAddIndexes
Testcase : testAddIndexesWithRollback(org.apache.lucene.index.TestAddIndexes):
  Caused an ERROR
    MockDirectoryWrapper: cannot close: there are still open files : {_co.cfs=1}
    java.lang.RuntimeException: MockDirectoryWrapper: cannot close: there are still
      open files : {_co.cfs=1}
      at org.apache.lucene.store.MockDirectoryWrapper.close(MockDirectoryWrapper.
        java:483)

```

<sup>30</sup><https://issues.apache.org/jira/browse/LUCENE-3251>

Table 5.4: Distribution of detection methods for memory and resource leaks.

		Mem. Leaks	Res. Leaks
Detection Type	Detection Method		
Source code-based detection	Manual code inspection	68 (28.6%)	113 (44.7%)
	Static analyzer	0 (0.0%)	1 (0.4%)
Runtime detection	Failed test	17 (7.1%)	40 (15.8%)
	Out-of-memory error	43 (18.1%)	12 (4.7%)
	Warning message	8 (3.4%)	11 (4.3%)
	Runtime (exclude above)	102 (42.9%)	76 (30.0%)

In some cases, the growth of memory usage leads to an out-of-memory (OOM) error during runtime. This is a severe symptom as the underlying system often crashes when an OOM error occurs. For example, DERBY-5730<sup>31</sup> reported a severe memory leak which might lead to a system crash due to an out-of-memory error. In this issue, the reporter mentioned that after removing the suspicious call, the test program was successfully executed with a much lower heap size. We should mention that we only consider those out-of-memory errors triggered via a resource or memory leak and not because of misconfigured heap size (which is a configuration error). The out-of-memory error due to a leak occurs at some point regardless of the heap size, while the OOM error due to a misconfiguration will not occur with correct configuration of the heap value. Logstash#5179<sup>32</sup> is an example of an OOM error due to a misconfiguration of the heap size for running a specific task which is fixed by setting the correct value for the heap size.

Developers also implement algorithms for the detection of specific leak defects. They usually warn the user about the potential presence of a leak with a message during the program’s execution. For example, in CXF-5707<sup>33</sup>, a message warned the user for a potential leak during the performance test of the *netty-http-server* module:

“SEVERE: LEAK: *ByteBuf.release()* was not called before it’s garbage-collected. Enable advanced leak reporting to find out where the leak occurred.”

<sup>31</sup><https://issues.apache.org/jira/browse/DERBY-5730>

<sup>32</sup><https://github.com/elastic/logstash/issues/5179>

<sup>33</sup><https://issues.apache.org/jira/browse/CXF-5707>

**Results.** Figure 5.4 shows the distribution of detection types in relation to the leak types. Table 5.4 illustrates the relationship between detection types, detection methods, and leak types.

**Finding 1.** More resource leaks (114 issues) are detected via source code-based detection than memory leaks (68 issues). Runtime detection is the dominant detection type for detecting memory leaks with 170 out of 238 issues (71.4% of the issues). The reason why more resource leaks are detected with source code-based detection techniques comes from the difference in memory and resource management. In Java, a programmer should explicitly dispose of the resources after usage. Due to explicit management, potential resource leaks can more often be captured through the code review or using static analyzers. Contrary to this, the JAVA Virtual Machine (JVM) manages the heap space and releases the unused objects when they become unreachable. Detecting unused references with code inspection is a hard task, as the programmer needs to have a profound understanding of the program’s workflow.

**Finding 2.** 309 (about 63%) issues are detected or manifest during the runtime. In these issues, users often use a third-party memory analyzer (e.g., *jmap*, *MAT*<sup>34</sup>, *yourkit*<sup>35</sup>), or OS-specific commands (e.g., *lsOf*) to verify the presence of the leaks. The information collected from these tools and commands can considerably help the developers to reproduce and diagnose the leak defects.

**Finding 3.** Users detected leaks in 19 issues (3.9%) via warning messages. From our dataset, we observe that in three applications, developers implemented leak detection mechanisms. This result shows that it is a good practice for developers to provide integrated leak detection mechanisms to accelerate the diagnosis of leak-related defects.

**Finding 4.** Out-of-memory errors are observed more than three times in memory leak-related issues. OOM error is one of the most severe leak symptoms and should be particularly prevented in a production environment.

**Finding 5.** In 57 (11.6%) issues, users detected the leaks via a test case. We also observe that for some issues, developers added a test case to the repair patches

---

<sup>34</sup><https://www.eclipse.org/mat/>

<sup>35</sup><https://www.yourkit.com/>

for future leak detection. This result shows the possibility of software tests as a lightweight tool for leak detection. Previous work [37, 41] confirm the effectiveness of software tests for leak detection. The utility of a test case is twofold. First, it can be used as an oracle for automated leak detection and bug isolation. Second, it can be an oracle for automated leak repair techniques as they need test cases to verify the correctness of their proposed fix patches. As leaks are highly sensitive to the environment (and input), the automated test input generation should provide inputs which can trigger the leaks in different execution paths.

**Finding 6.** Only in one issue (CASSANDRA-7709<sup>36</sup>), the leak is detected and reported by a static analyzer. As we only consider the reported issues, we cannot generalize that the static analysis tools are not used. It is possible that static analyzers have been employed earlier in the development process, and all leaks detected were fixed. Still, our finding highlights potentials to improve the existing static analyzers further as there is still room for improvement. It is essential to know why these tools are not used for other reported issues with similar characteristics to the detected issue (we further analyze this in Section 5.4.7). One reason might be that there are still obstacles in the extensive use of such debugging tools. Such obstacles can be high false positives, complicated usage procedure, or lack of knowledge about these tools. Researchers or tool builders should improve current debugging tools to detect not-yet covered bugs, simplify the tool usage, and spread them widely in the community.

**Summary.** Source code-based detection is more common in resource leak detection (45.1%). Runtime detection is the dominant detection type for memory leaks (71.4%). Out-of-memory errors are observed about three times more frequently in conjunction with the memory leaks than with the resource leaks.

### 5.4.3 RQ3: To What Extent Are the Leak-Inducing Defects Localized?

**Motivation.** Fault localization is the first step of bug diagnosis. The locality of a fault can be defined in different granularity such as statement, method, and file. In the

<sup>36</sup><https://issues.apache.org/jira/browse/CASSANDRA-7709>

## 5. An Empirical Study on Leak-inducing Defects and Their Repairs

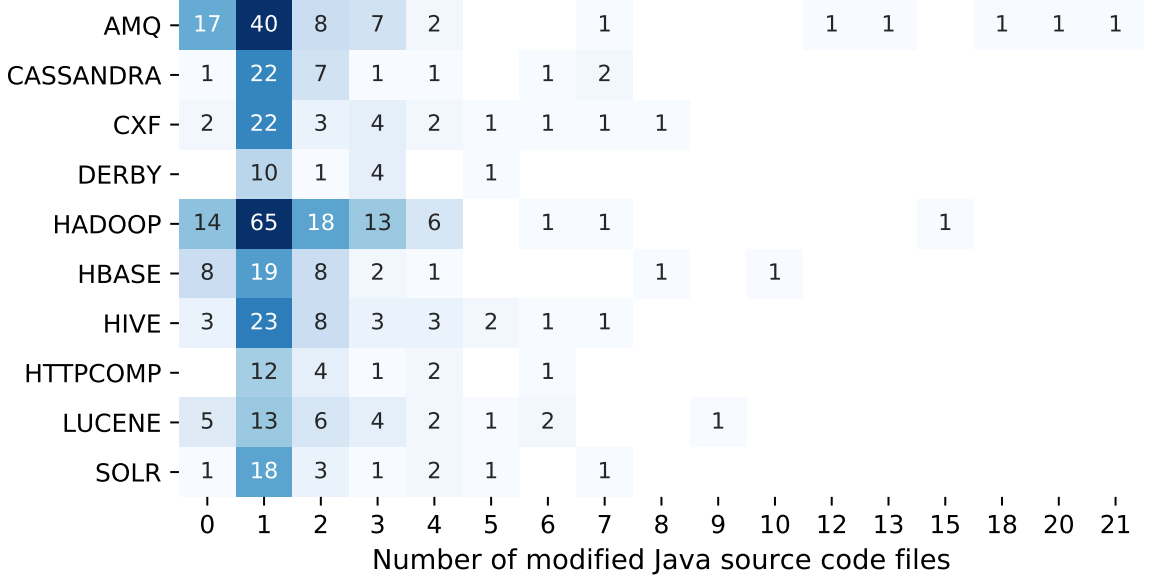


Figure 5.5: Heatmap of the number of modified Java source code files per project.

case of leak-related defects, they can affect a region (e.g., multiple modules, classes) in the codebase of an application [88]. Accurate defect localization is vital in providing the correct repair patch. Otherwise, the patch will not fix the bug completely and even introduces a new bug [135]. In this research question, we investigate the locality of leak-inducing defects. In particular, we want to find out: (1) how many source code files are changed to repair a defect, and (2) which types of files are changed in each repair patch.

**Approach.** To assess the locality of leak defects, we analyze the distribution of modified source code files. For each issue, we collect the files changed in the repair patches. We also investigate the file type of modified source files by collecting the file extensions. We ignore test files in the repair patches if the tests added or modified for future leak detection and not for repairing purposes.

**Results.** Figure 5.5 shows the heatmap representation of the amount of modified Java source files for each project.

**Finding 1.** In 57% of the issues, developers changed only one source code file to repair the defect. In about 81% of the issues, three source code files are modified at most. This result implies the high locality of leak-inducing defects considering file-level granularity.



**Finding 2.** In 15 issues, developers repaired the defect via adding or deleting at least one Java source code file. It is interesting to know the reasoning behind such changes. However, the information for such decisions is not always provided in the issue reports and reasoning require a deep knowledge about the design and architecture of the project. Our further analysis shows that the decision of adding or deleting a class is simply a design decision and it is very particular to the issue being fixed. Here, we provide three examples of such cases. In CASSANDRA-552<sup>37</sup> developer created a new interface (in a new file) which makes an iterator object to be closable. In HADOOP-639<sup>38</sup> developers redesigned the code with unifying two existing functionalities. In LUCENE-1383<sup>39</sup> developers implemented a closable Java `ThreadLocal` as a wrapper for Java `ThreadLocal` which wraps the values inside a `WeakReference` with keeping a strong reference to the value. In this way, the garbage collector does not reclaim the value until the close method is called. In general, we could not generalize any specific rule about adding or deleting files in the repair patches of the studied issues.

Albeit occurring only in about 3% of the studied issues, such cases require more sophisticated repair strategies. Most of the current automated program repair approaches [62, 66, 123] can only provide simple repair patches with only one line. Hence, it is still not feasible for existing automated program repair techniques to provide complex repair patches such as adding a class.

**Finding 3.** In 17 issues, no Java source code file is changed. In eight issues, source code files written in C are modified. In three cases, developers modified the XML files to use a non-leaky third-party library as a dependency for that project. Six issues are repaired by changing source code files written in Scala and Ruby. The reason for changing different file types is that in some of the studied projects, specific modules are implemented in different programming languages than Java. For example, `bzip2` (a compression method) implementation in HADOOP is written in C.

Test cases might also contain a leak in their code. For example, YARN-2662 reports an issue where the tests do not close a file after reading from it. We observe 15 issues in our dataset that the repair patches contain only changes in the test files.

---

<sup>37</sup><https://jira.apache.org/jira/browse/CASSANDRA-552>

<sup>38</sup><https://jira.apache.org/jira/browse/HADOOP-639>

<sup>39</sup><https://jira.apache.org/jira/browse/LUCENE-1383>

## 5. An Empirical Study on Leak-inducing Defects and Their Repairs

A bug report is labeled as duplicated if it has already been reported in the bug tracker. However, it can be the case that a reporter reports a bug and this bug is already repaired in one of the previous releases of the software, or the root cause of the leak is located in a third-party library. In such cases, developers close the issue as fixed with referring to the software release containing the bug fix or the non-leaky third-party library. In our study, we find 29 issues of such cases, i.e., the issues are closed without including a repair patch. It is worth mentioning that these issues cannot be considered as duplicated because they are not previously been reported in the bug tracker (i.e., there is no link to another issue in the bug tracker).

Although only a few defects are repaired by modifying files written in other programming languages, developers require knowledge of different programming languages to repair all leak-related defects.

**Summary.** About 54% of the leak defects are repaired by changing only one source code file. Only in 12% of the reported leaks, more than three source files were modified. In about 6% of the issues, files from other languages, such as C, Scala, and Ruby are modified to fix the leak-related defect.

### 5.4.4 RQ4: What Are the Most Common Root Causes?

**Motivation.** In this research question, we want to find out which root causes are dominant, and whether there are significant differences in the common root causes for different leak types.

**Approach.** To find the root cause, we use three data sources for each issue: issue title, issue description, and developers discussions. The categories for root cause are not mutually exclusive. For issues with the possibility of assignment to multiple categories, we select the most specific category as explained in Section 5.3.5.

**Taxonomy of defect types.** Table 5.5 lists the taxonomy of the defect types. We describe the most common root causes here.

**Non-closed resource.** The programmer should close any system resources such as file handles, connections, and threads when they are no longer needed. Otherwise, a

resource leak is likely. For example, in HBASE-12837<sup>40</sup>, zookeeper connections created in the constructor of `ReplicationAdmin` left unclosed.

*Bad exception handling.* According to Java documentation<sup>41</sup>, an exception is an event which disrupts the normal flow of the program’s instructions. When an exception is thrown, any resources accessed during the normal execution of the program remain open. If a programmer does not properly handle the exceptions, a leak is prone to happen, as shown in the following quote from an issue report:

“Programmer should handle the exception properly instead of swallowing it.”

For instance, in LUCENE-3144<sup>42</sup>, `FreqProxTermsWriter` leaks open file handle if an exception is thrown during `flush()`.

*Collection mismanagement.* The mismanagement of elements in a collection can result in a memory leak. Such leak occurs when a programmer assumes that the garbage collector collects all unused objects, even if they are still referenced. Leaks due to collection mismanagement can lead to severe memory waste, in particular when the collection is used as a static member. The reason is that the static fields are never garbage-collected. Issue YARN-5353<sup>43</sup> reports a severe memory leak due to keeping the tokens in the `appToken` map of the `ResourceManager` even after task completion.

*Concurrency defect.* A leak can be caused by a race condition preventing the disposal of a resource or releasing references to unused objects. Issue LUCENE-6499<sup>44</sup> reports a file handle leak if files are concurrently opened and deleted.

**Results.** We investigate the frequency of the root causes across the leak types. Table 5.5 and Figure 5.6 summarize the results. Table 5.5 lists the common root causes and their corresponding number of issues. Figure 5.6 visualizes the heatmap of the defect and leak types.

**Finding 1.** The majority of the defects (about 76% of the cases) manifest when a normal execution path is exercised. The most common root cause is also the non-

<sup>40</sup><https://issues.apache.org/jira/browse/HBASE-12837>

<sup>41</sup><https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>

<sup>42</sup><https://issues.apache.org/jira/browse/LUCENE-3144>

<sup>43</sup><https://issues.apache.org/jira/browse/YARN-5353>

<sup>44</sup><https://issues.apache.org/jira/browse/LUCENE-6499>

Table 5.5: Taxonomy of root causes. Column “#Issues” states the total number of issues per root cause.

Description (Short)	#Issues
Non-closed resource at error-free execution ( <b>nonClose</b> )	149 (32.96%)
Object not disposed of if exception is thrown ( <b>excepti</b> )	98 (21.68%)
Dead objects referenced by a collection ( <b>collection</b> )	93 (20.58%)
Unreleased reference at error-free execution ( <b>unreleas</b> )	59 (13.05%)
A race condition defect ( <b>concurrency</b> )	18 (3.98%)
Wrong call schedule of disposal method ( <b>callSchedule</b> )	16 (3.54%)
Over-sized cache or buffer ( <b>cache</b> )	14 (3.10%)
Incorrect API usage ( <b>wrongAPI</b> )	10 (2.21%)
Unreleased reference due to thread-local variable ( <b>thre</b> )	10 (2.21%)
ClassLoader keeps a bi-directional reference to a class (	10 (2.21%)
Leaks related to Java native interface ( <b>jni</b> )	8 (1.77%)
Leak inside a third-party library ( <b>leakyLib</b> )	7 (1.55%)

closed resource in a regular (error-free) execution path (*nonClosedRes*) with about 30% of the cases. This finding is interesting. The error-free execution paths are more often executed and checked. Therefore, it should be less likely for defects to manifest in normal execution paths [122]. However, our analysis shows that this is not the case for leak-related defects. Our analysis shows the value of software tools and tests which check whether resources are disposed of at the end of the typical execution paths.

**Finding 2.** Bad exception handling (*exception*) is the second-most frequent root cause with 20% of the issues (93 issues). This even increases to about 32% of the issues if we only consider resource leaks. We also observe that this root cause is about five times more common for resource leaks than for memory leaks. One reason for such observation is that - by definition - exception paths happen in exceptional situations, being less frequently tested than normal execution paths. Even correctly-behaved programs in normal execution path can manifest error in exceptional paths [121, 122]. This observation implies that the proper exception handling plays an important role in preventing leaks especially resource leaks.



Figure 5.6: Heatmap of defect types and leak types.

**Finding 3.** Collection mismanagement (*collection*) is the most common root cause for memory leaks (39% of the cases). This finding verifies the applicability of existing automated approaches for detecting memory leaks caused by collection mismanagement (e.g., [131]).

**Summary.** Four types of defects cause most leaks. Collection mismanagement (39% of the issues) and non-closed resources (58% of the issues) are the dominant root causes for memory and resource leaks, respectively. The majority of the leaks (76% of the cases) manifest in the normal execution paths.

#### 5.4.5 RQ5: What Are the Characteristics of the Repair Patches?

**Motivation.** One approach for automated program repair is to search for common repairs from previous fix patches and provide repair candidates to fix bugs [62, 66,

74, 105, 110]. Align with this direction we investigate the repair patches to check whether there are common patterns for fixing the leak-inducing defects.

**Approach.** For each issue in our dataset, we manually check the issue summary, the issue description, the developer discussions, and the repair patches to understand the design philosophy of a fix and find the repair action and corresponding code transformation for each defect. A repair action corresponds to an abstract description of a fix, while a code transformation is a concrete instantiation of the repair action. The same abstract fix can be implemented via different code transformations. For example, to fix a leak due to a non-closed resource (a defect type), the developer needs to dispose of the resource (a repair action). Disposing of a resource can be implemented using a `try-finally` construct (a possible code transformation) or a `try-with-resources` construct (another possible code transformation).

When analyzing the patches, we apply the following considerations. First, we are only interested in the defects within the codebase of the application. Hence, we ignore modifications of the test files in the repair patches. Second, in 29 issues, the defects are already repaired by developers in another version of the application but were not tagged as “duplicate” in the bug tracker. We ignore these issues for searching for common code transformation. Every label is attributed to a specific repair action whenever possible. We categorize the fix patch to a general category only when no specific repair action would fit the repair description.

To identify the common code transformations that may be applied for fixing multiple issues, we use the open coding methodology. First, we label each repair patch with all code transformations associated with that patch. Then, we identify those common transformations that repeatedly occur (more than once) within our dataset.

**Taxonomy of repair actions.** Table 5.6 lists the repair actions. Note that the repair actions are mutually exclusive. For issues with the possibility of assignment to multiple categories, we select the most specific category as explained in Section 5.3.5. We describe the common actions here.

*Dispose of resource on a regular path (disposeReg).* Resource leak defects introduced in *regular* execution paths can be resolved via directly calling the disposal method after the resource usage. In Java, this is achieved by calling the `close` or `dispose`

Table 5.6: Taxonomy of repair actions. Column “#Issues” states the total number of issues per repair action.

Description(Short)	#Issues
R1: Dispose of resource in regular execution paths ( <b>dispose</b> )	111 (24.56%)
R2: Dispose of objects in exceptional path ( <b>disposeE</b> )	97 (21.46%)
R3: Remove the elements from a collection ( <b>removeE</b> )	104 (23.01%)
R4: Release the reference ( <b>releaseRef</b> )	69 (15.27%)
R5: Shutdown thread after finishing the task ( <b>thread</b> )	45 (9.96%)
R6: Improve thread safety by avoiding race condition ( <b>thread</b> )	23 (5.09%)
R7: Use an efficient API to improve memory usage ( <b>cc</b> )	12 (2.65%)
R8: Modify strong reference to a weak reference ( <b>weak</b> )	9 (1.99%)
R9: Use a non-leaky Library ( <b>nonLeakyLib</b> )	4 (0.88%)
R10: Bugs not belonging to the above categories ( <b>other</b> )	17 (3.76%)

method. For example, in HADOOP-7090<sup>45</sup>, the developer refers to closing the I/O streams in a `finally` block as a *good practice*. Following is a partial patch for this issue:

```

--- org/apache/hadoop/io/BloomMapFile.java
+++ org/apache/hadoop/io/BloomMapFile.java
@@ -186,10 +186,17 @@
    @Override
    public synchronized void close() throws IOException {
        super.close();
-       DataOutputStream out = fs.create(new Path(dir, BLOOM_FILE_NAME), true);
-       bloomFilter.write(out);
-       out.flush();
-       out.close();
+       DataOutputStream out = null;
+       try {
+           out = fs.create(new Path(dir, BLOOM_FILE_NAME), true);
+           bloomFilter.write(out);

```

<sup>45</sup><https://issues.apache.org/jira/browse/HADOOP-7090>

```
+    out.flush();
+    out.close();
+    out = null;
+    } finally {
+        IOUtils.closeStream(out);
+    }
```

*Release reference.* Any unused object in Java should be reclaimed by GC. If this object is still reachable by a live object, GC will not release its memory footprint. In such cases, the responsibility lies on the programmer to release the references preventing the object from being garbage collected (e.g., by nullifying the references to the unused objects). For example, HBASE-5141<sup>46</sup> reports a memory leak due to keeping references, even the corresponding task is finished. The fix patch nullifies the no-longer-needed objects. Following is the partial patch:

```
--- org/apache/hadoop/hbase/monitoring/MonitoredRPCHandlerImpl.java
+++ org/apache/hadoop/hbase/monitoring/MonitoredRPCHandlerImpl.java
@@ -217,6 +217,13 @@
...
+ @Override
+ public void markComplete(String status) {
+     super.markComplete(status);
+     this.params = null;
+     this.packet = null;
+ }
+
```

*Proper exception handling (disposeExcp).* Programmer should dispose of the objects or resources in all *exceptional* execution paths. Otherwise, a leak is likely to happen when an exception is thrown. Issue AMQ-3052<sup>47</sup> reports a memory leak in securityContexts. It occurs when the addConnection() fails after a successful authentication check. The developer fixed the bug via adding a try-catch block and calling disposal method in the catch block:

<sup>46</sup><https://issues.apache.org/jira/browse/HBASE-5141>

<sup>47</sup><https://issues.apache.org/jira/browse/AMQ-3052>



```

--- org/apache/activemq/security/SimpleAuthenticationBroker.java
+++ org/apache/activemq/security/SimpleAuthenticationBroker.java
@@ -92,7 +92,13 @@
...
- super.addConnection(context, info);
+ try {
+     super.addConnection(context, info);
+ } catch (Exception e) {
+     securityContexts.remove(s);
+     context.setSecurityContext(null);
+     throw e;
+ }

```

*Remove an element from a collection (removeElm).* No longer needed members of a collection should be removed by the programmer, allowing the garbage collector to reclaim the memory. A typical repair action is the call of *remove()* method of a collection to clear useless elements from being referenced by the collection object. For example, in issue YARN-3472<sup>48</sup>, already expired and removed tokens are not removed from `allTokens` map resulting in a potential memory leak. Developer fixed the issue by adding a call to `remove` method which removed the expired token from the map.

```

--- /hadoop/yarn/server/resourcemanager/security/DelegationTokenRenewer.java
+++ /hadoop/yarn/server/resourcemanager/security/DelegationTokenRenewer.java
@@ -577,6 +577,7 @@ private void requestNewHdfsDelegationTokenIfNeeded(
...
    if (t.token.getKind().equals(new Text("HDFS_DELEGATION_TOKEN"))) {
        iter.remove();
+       allTokens.remove(t.token);
...

```

*Shutdown finished thread (threadDown).* A live thread of the application should be destroyed by the programmer when the thread task is finished. Adding a call to the `shutdown` method or adding a disposal method are the common repair actions

<sup>48</sup><https://issues.apache.org/jira/browse/YARN-3472>

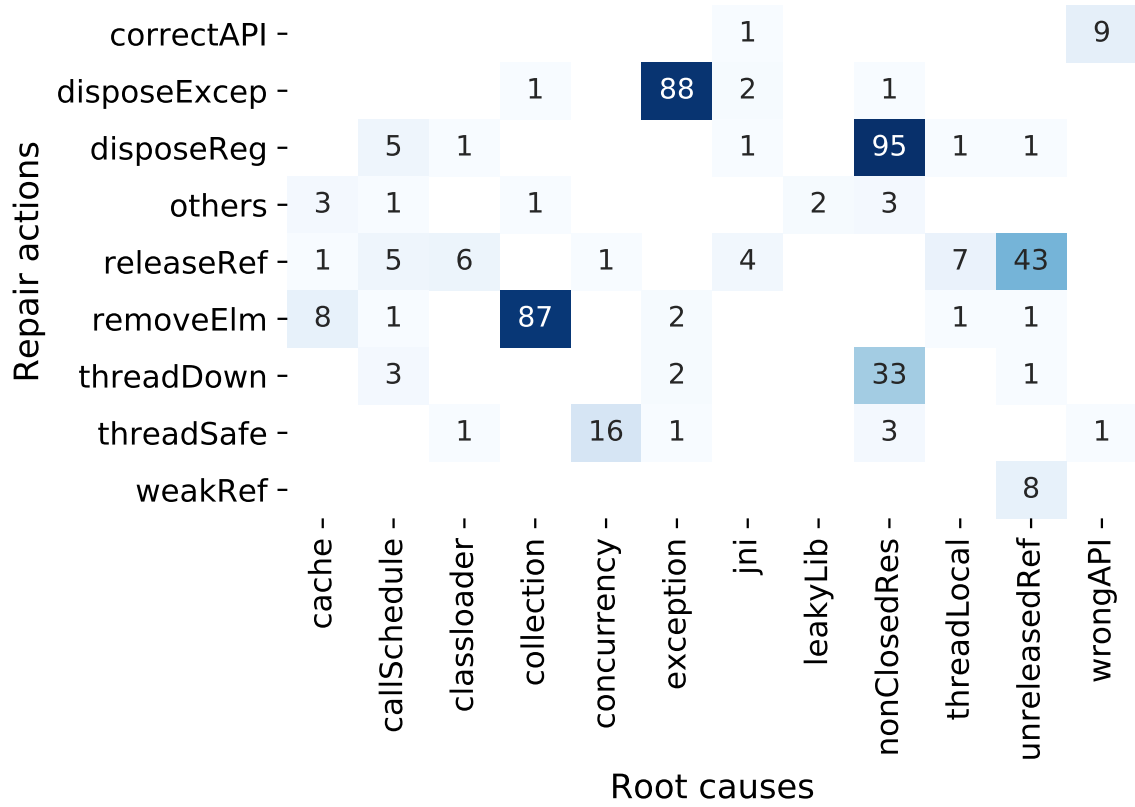


Figure 5.7: Heatmap of relationship between root causes and repair actions.

for fixing the leaks caused by threads. HDFS-9003<sup>49</sup> reports a thread leak when a standby NameNode initializes the quota. Here, the thread pool is not shut down. To fix this bug, the developers added a call to the shutdown method.

```

--- org/apache/hadoop/hdfs/server/namenode/FSImage.java
+++ org/apache/hadoop/hdfs/server/namenode/FSImage.java
@@ -880,6 +880,7 @@ static void updateCountForQuota(BlockStoragePolicySuite bsp,
    root, counts);
    p.execute(task);
    task.join();
+   p.shutdown();

```

**Results.** In following, we show the results of our analysis on the repair patches in three sub-questions. First, we study the frequency of the repair actions. Second,

<sup>49</sup><https://issues.apache.org/jira/browse/HDFS-9003>

Table 5.7: Recurring code transformations and examples of code before and after the transformations.

**Code transformation 1:** Conditional disposal of resource.

**Example:** `dispose(obj) → If (obj != null) obj.dispose()`

**Code transformation 2:** Add disposal method call.

**Example:** `None → obj.dispose()`

**Code transformation 3:** Add disposal method.

**Example:** `None → void dispose()`

**Code transformation 4:** Set obsolete reference to null.

**Example:** `None → obj=null`

**Code transformation 5:** Add catch/try-catch block.

**Example:** `Type obj = new Type() →  
try {Type obj = new Type()} exception {dispose(obj)}`

**Code transformation 6:** Add finally/try-finally block

**Example:** `Type obj = new Type() →  
try {Type obj = new Type()} finally {dispose(obj)}`

**Code transformation 7:** Add try-with-resources statement.

**Example:** `Type obj = new Type() → try {Type obj = new Type() }`

**Code transformation 8:** Change condition expression.

**Example:** `If (cond1) obj.dispose() → If (cond1 and cond2) obj.dispose()`

**Code transformation 9:** Change method call parameters.

**Example:** `obj.method(x, y) → obj.method(x, z)`

**Code transformation 10:** Change static object to a non-static.

**Example:** `static Type obj = new Type() → Type obj = new Type()`

**Code transformation 11:** Change to weak reference.

**Example:** `new HashMap<key, value>() →  
new HashMap<key,WeakReference(value)>()`

**Code transformation 12:** Replace method call.

**Example:** `obj.method1() → obj.method2()`

**Code transformation 13:** Change collection.

**Example:** `obj = new <collection1>() → obj = new <collection2>()`

we analyze the mapping between the root causes and the repair actions to find the relationship between these two taxonomies. Finally, we report the common code transformations found in the fix patches.

**Finding 1.** Table 5.6 lists the common repair actions along with the number of issues corresponding to them. About 93% of the resource leaks are repaired with three major actions: *disposeReg*, *disposeExcep*, and *threadDown*, while about 73% of the memory leaks are fixed with two repair actions *releaseRef* and *removeElm*.

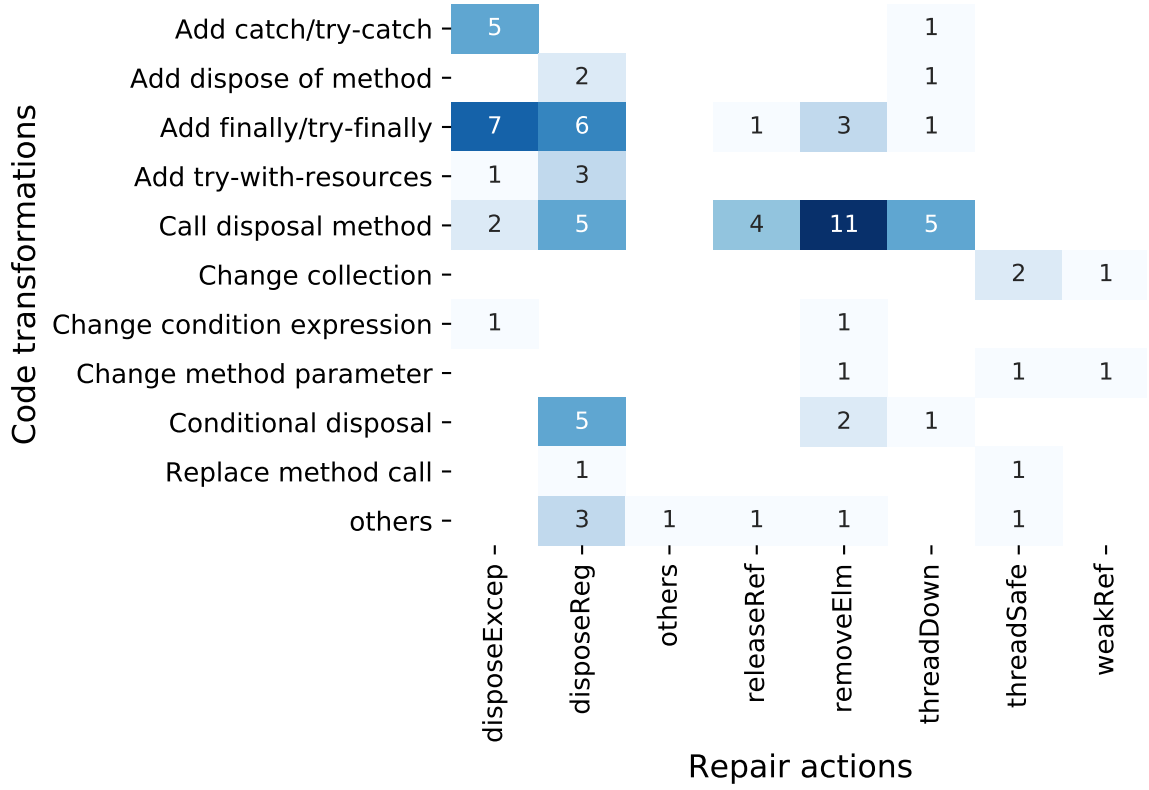


Figure 5.8: Heatmap of recurring code transformations and single repair actions.

**Finding 2.** Figure 5.7 reveals an almost one-to-one mapping between some root causes and repair actions (e.g., *exception* → *disposeExcep*, *collection* → *removeElm*, *concurrency* → *threadSafe*, *concurrency* → *threadSafe*). Leak defects with the root cause type *nonClosedRes* are repaired with repair actions *threadDown* and *disposeReg*. Leak defects with the root cause type *unreleaseRef* are repaired with repair actions *releaseRef* and *weakRef*.

**Finding 3.** We find 13 recurring patterns in the repair patches. Table 5.7 lists the recurring code transformations and the code examples before and after the transformations. Our analysis shows that 88 (out of 491) issues are repaired with a single code transformation. For these issues, we further analyze the quantitative relationship between the repair actions and the most common code transformations. Figure 5.8 shows the heatmap of the quantitative analysis. For this heatmap, we only consider repair patches with a single code transformation. Code transformation *Add finally/try-finally* is often used in the repair actions *disposeReg*

or *disposeExcep*. Code transformation *Add catch/try-catch* is the most used code transformation for repair action *disposeExcep*. We also observe a direct relationship between the repair action *RemoveElm* and the code transformation *Call disposal method*.

This result can encourage the researchers to implement patches for leak-related defects based on template-driven patch generation techniques in the direction of previous work [62, 85, 97].

**Summary.** Overall, five repair actions are used by developers to repair over 86% of the issues in our dataset. We found 13 recurring code transformations. 88 of the issues are repaired with a single code transformation.

#### 5.4.6 RQ6: How Complex Are Repairs of the Leak-Inducing Defects?

**Motivation.** This research question addresses the complexity of changes applied to repair the leak-inducing defects. Besides this, we analyze the time to resolve (TTR) for different repair actions. We also compare TTR between leak-related and non-leak-related defects. In this question, we want to find how complex are the repair patches. The results can provide more insights on how complex are the repair patches and how long does it take to repair a leak-inducing defect.

**Approach.** To assess the complexity of changes, we compute the code churn and change entropy [48].

Code churn is the sum of added and deleted lines in a repair patch. We only consider changes in the code statements and ignore comments or empty lines when calculating the code churn metric.

We use change entropy to find scatteredness of the changes. Derived from Shannon entropy in information theory, the change entropy measures the complexity of the changes. To measure the change entropy, we use the normalized Shannon entropy [24, 48]. It is defined as:

$$H = \frac{-\sum_{i=1}^n p(file_i) \cdot \log_e p(file_i)}{\log_e(n)},$$

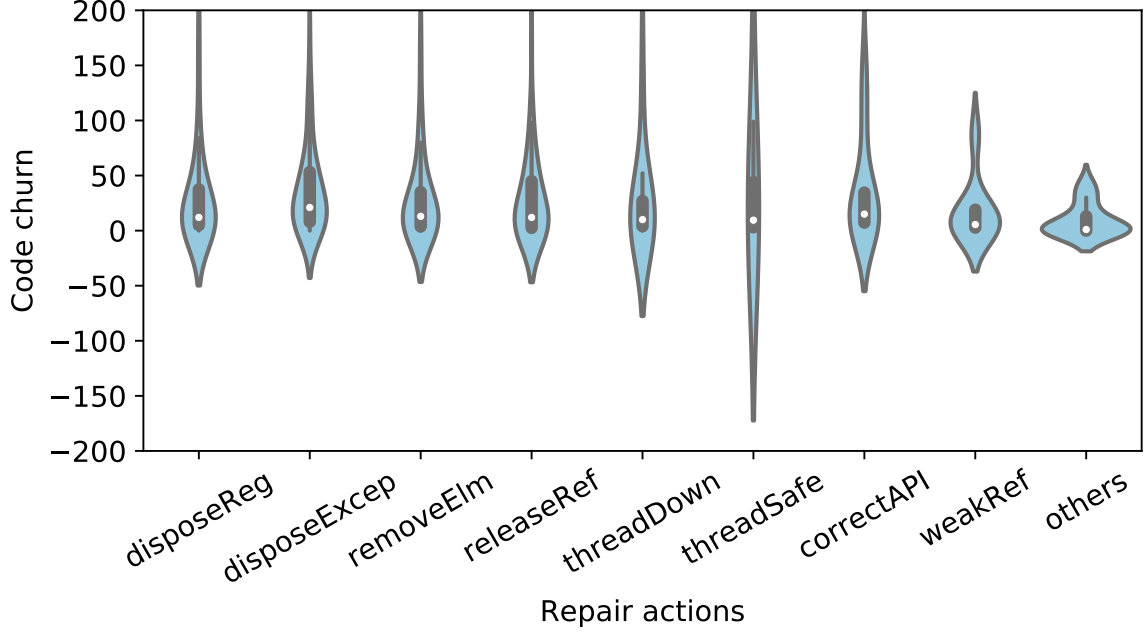


Figure 5.9: Distribution of code churn per repair action.

where  $n$  is the total number of files in a repair patch and  $p(file_i)$  is defined as the number of lines changed in  $file_i$  over the total number of lines changed in every file of that repair patch. Change entropy achieves its maximum value when all the files in a repair patch have the same number of modified lines.

In contrast, we can achieve a minimum of entropy when only one file has the total number of modified lines. Using the entropy, we can find how complex are the repair patches. The higher the entropy, the more complex the repair patch.

To assess the time to resolve (TTR) of an issue report, we adopted the methodology used in previous studies [57, 92, 110]. We collect two timestamps from each issue report: the time it was created (recorded in the issue tracker), and the time it was resolved (labeled as resolved). For GitHub projects, we use the closed timestamp as the resolved timestamp as it is the only available timestamp in the issue report. For issues with multiple patches, the resolved timestamp is the time of the latest patch being applied to repair the bug. The TTR is the difference between created and resolved timestamps. The reason for using TTR is that the bug trackers used in this study (i.e., Jira and GitHub bug tracker) record no information about the exact amount of coding time needed for fixing a bug.

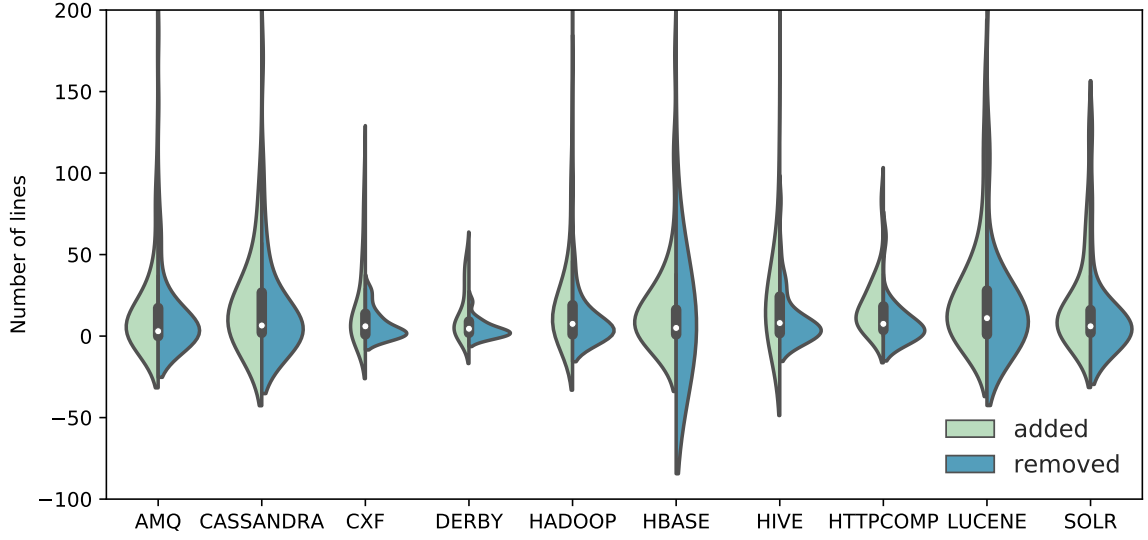


Figure 5.10: Distribution of number of added and removed lines over the studied projects.

**Results.** In the following, we show the results of our analysis of the complexity of the repair patches.

*Distribution of code churn..* Figure 5.9 shows the box plot of code churn for different repair actions. The line within each box points to the median value of the code churn for that repair action.

**Finding 1.** In about all repair actions, the median of code churn is less than 20 lines of code while the repair action *disposeExcp* shows the highest median value.

**Finding 2.** Figure 5.10 shows the distribution of added and removed lines over studied projects. In all the projects, the median of added lines (29.5 lines) shows a higher value than the removed lines (16.5 lines). Hence, the fault repairing changes often increase the codebase of the applications.

**Finding 3.** Figure 5.11 shows the distribution of change complexity over the repair patches. The distribution appears to be bimodal with the main peak around zero and a lower one around one. The change complexity analysis shows that the changes applied for repairing leak-inducing defects are more concentrated in fewer files and are less scattered. This result can be a useful finding for automated fault localization as it shows the high localization in leak-inducing defects.

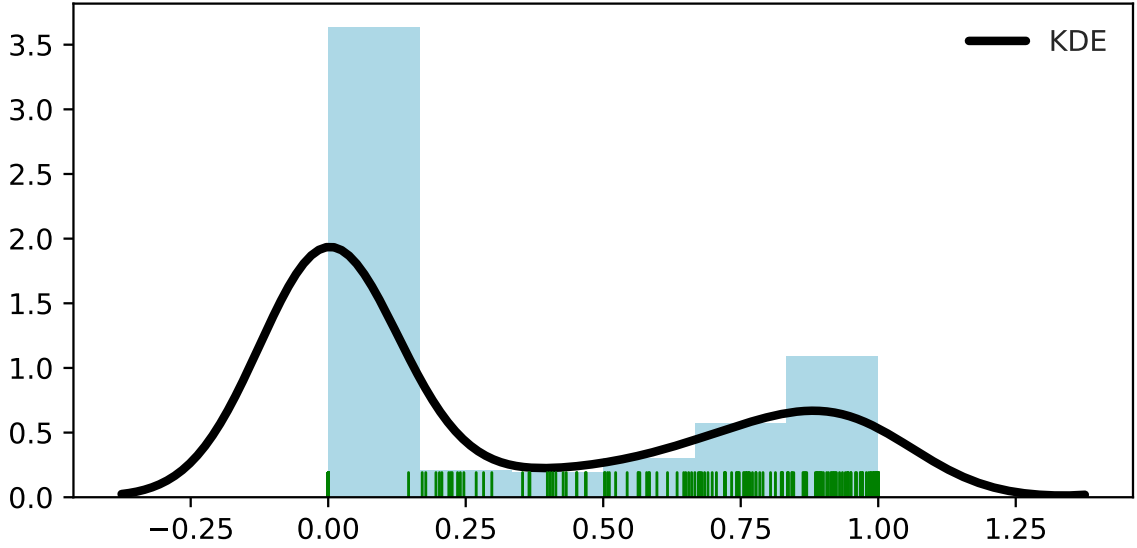


Figure 5.11: Distribution of change complexity over the repair patches.

*Time to resolve (TTR)*. Figure 5.12 shows distribution of TTR across repair actions. Figure 5.13 shows the distribution of the TTR for the leak-related and other defects in the studied projects. To calculate the TTR for other defects, we collect the created and resolved timestamps of all bugs with the resolution “FIXED” (except the leak-related defects) from the studied projects in the same time frame that we collected the leak defects.

**Finding 4.** On a median, about 5.88 days is needed for developers to fix a leak-inducing defect. This time is slightly lower than the TTR for repairing non-leak defects (about 6.04 days). One reason could be that leak-related defects are important for users and developers. The data in our dataset also confirms this. The issue priority in about 84% of the issues in projects from Apache are labeled as *Blocker*, *Critical*, or *Major* (which are the highest priority levels in the Jira bug tracker). This corroborates with the assumption that leak-inducing defects impose a high negative impact on the performance of the applications, and are highly prioritized by development teams.

**Summary.** The change entropy shows that the changes are more concentrated in fewer files and therefore less scattered. The median TTR of the leak-inducing defects is about 5.88 days.



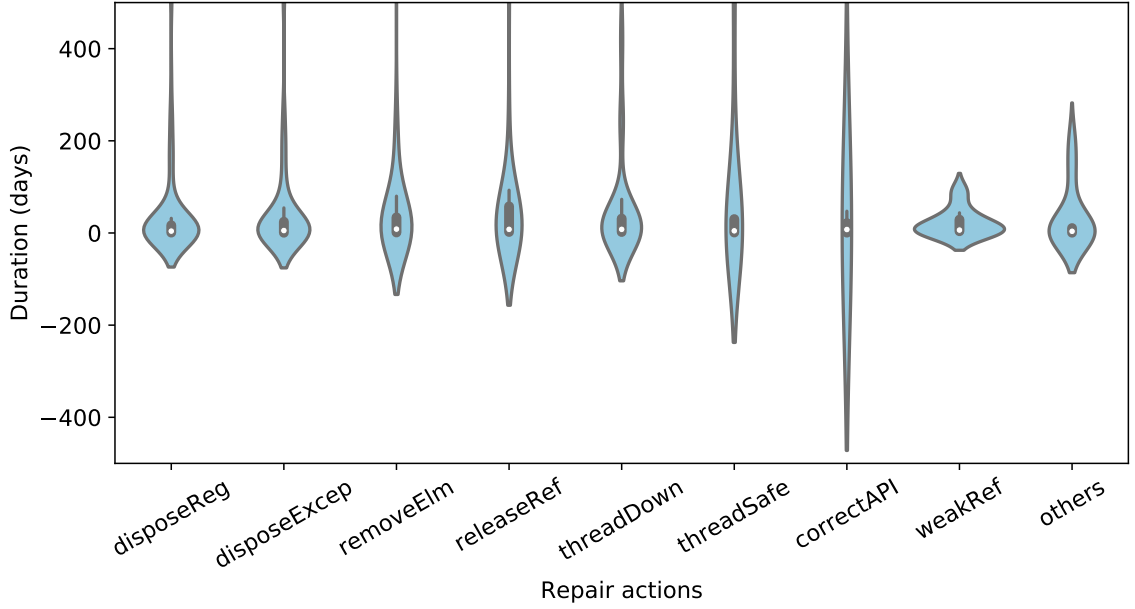


Figure 5.12: Distribution of time to resolve per repair action.

### 5.4.7 Other Findings

In this section, we provide other findings found by our study.

**Efficiency of static analysis tools.** In RQ2 (Section 5.4.2), we showed that only in one issue (i.e., CASSANDRA-7709), the resource leak was reported using a static analyzer. There are many static analysis tools for leak detection. They are mostly used for resource leak detection (e.g., FindBugs, Coverity, and Infer). Static analyzers can also be used to detect memory leaks. However, static memory leak detection is imprecise and not scalable for large programs [130, 132]. This inefficiency can be attributed mainly to the presence of the garbage collector and lack of runtime information. However, one could ask why these tools are not mentioned in the studied issue reports. One reason might be that static analyzers have been employed earlier in the development process, and all leaks detected were fixed.

In our study, we showed that more than half of the studied leaks are resource leaks. It is interesting to study whether static analyzers can detect the studied leak issues. For this purpose, we evaluate our dataset. We randomly select 30 issues reporting resource leaks from our dataset. As a static analysis tool, we choose Infer which is

Table 5.8: The evaluation of Infer static analyzer on a sample of resource leaks from our dataset. Column “Det?” reports whether Infer could detect the defect reported in the respective issue. “Code-based detection” refers to source code-based detection. “Defect” type and “Repair” type are explained in Section 5.4.4 and Section 5.4.5, respectively.

Issue	Det?	Detection	Defect	Repair
<b>AMQ-5745</b>	✓	<b>Code-based</b>	<b>nonClosedRes</b>	<b>disposeReg</b>
AMQ-6051	No	Runtime	exception	disposeExcep
CASSANDRA-7709	No	Runtime	exception	disposeExcep
CASSANDRA-9134	No	Runtime	nonClosedRes	disposeReg
<b>DERBY-5480</b>	✓	<b>Runtime</b>	<b>nonClosedRes</b>	<b>disposeReg</b>
HADOOP-10203	No	Runtime	nonClosedRes	disposeReg
<b>HADOOP-10490</b>	✓	<b>Runtime</b>	<b>nonClosedRes</b>	<b>disposeReg</b>
HADOOP-11014	No	Code-based	exception	disposeExcep
HADOOP-11056	No	Code-based	exception	disposeExcep
HADOOP-11349	No	Code-based	exception	disposeExcep
HADOOP-11368	No	Runtime	nonClosedRes	threadDown
HADOOP-11414	No	Code-based	exception	disposeExcep
<b>HADOOP-9681</b>	✓	<b>Runtime</b>	<b>nonClosedRes</b>	<b>disposeReg</b>
HBASE-10461	No	Code-based	exception	disposeExcep
<b>HBASE-10995</b>	✓	<b>Code-based</b>	<b>nonClosedRes</b>	<b>disposeReg</b>
HBASE-13601	No	Runtime	exception	disposeExcep
<b>HBASE-13797</b>	✓	<b>Code-based</b>	<b>nonClosedRes</b>	<b>disposeReg</b>
HDFS-1118	No	Code-based	exception	disposeExcep
HDFS-1753	No	Code-based	exception	disposeExcep
HDFS-5099	No	Runtime	nonClosedRes	disposeReg
HDFS-5671	No	Runtime	exception	disposeExcep
HDFS-6208	No	Code-based	nonClosedRes	disposeReg
<b>HDFS-6238</b>	✓	<b>Runtime</b>	<b>nonClosedRes</b>	<b>disposeReg</b>
HIVE-12250	No	Runtime	nonClosedRes	disposeReg
HIVE-12790	No	Runtime	nonClosedRes	disposeReg
HIVE-13405	No	Code-based	exception	disposeExcep
MAPREDUCE-6528	No	Runtime	exception	disposeExcep
YARN-2484	No	Code-based	exception	disposeExcep
<b>YARN-2988</b>	✓	<b>Code-based</b>	<b>nonClosedRes</b>	<b>disposeReg</b>
YARN-4581	No	Runtime	exception	disposeExcep

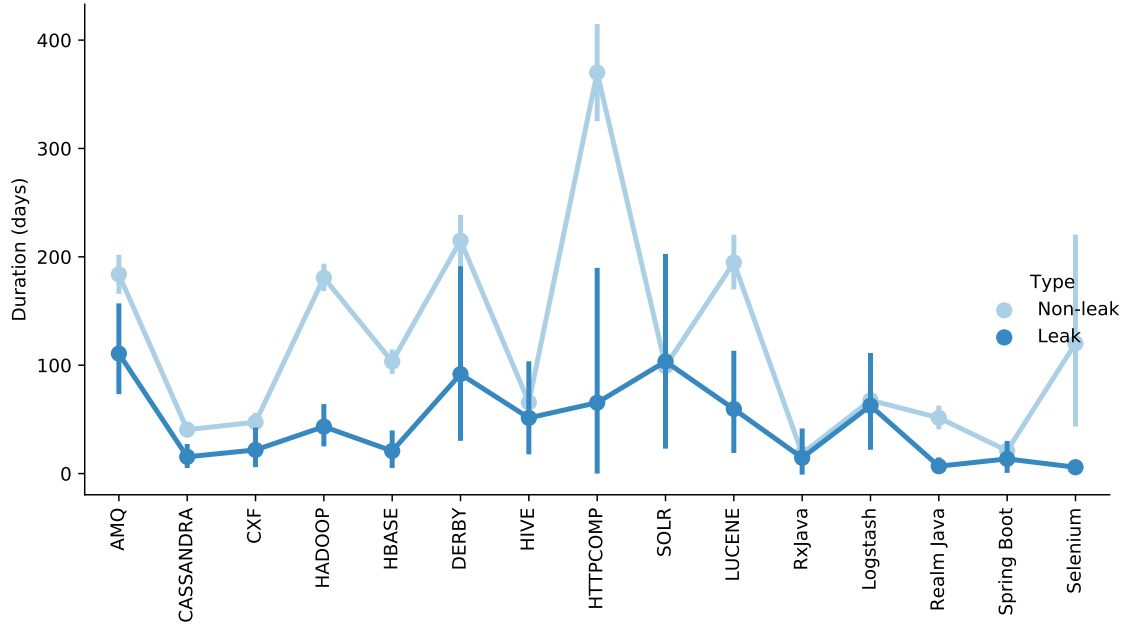


Figure 5.13: TTR comparison of leak-related and other bugs in studied projects.

used by large software organizations<sup>50</sup>. We selected Infer because it is an open source tool and can detect resource leaks in Java. The applicability of Infer for resource leak detection is also confirmed in the previous work [118].

Table 5.8 shows the result of our evaluation. From 30 issues, Infer was able to detect the leak defects reported in eight issues. To apply Infer, we first have to build the buggy version of the application in question which contains the leak. After a successful build, Infer produces a file reporting all potential resource leaks. Finally, we investigate whether the file contains the reported leak. We further investigate the eight issues detected by Infer to find the shared characteristics among those issues. In all cases, the leaks occurred in normal execution paths. The analysis shows that Infer was not able to detect leaks triggered in exceptional paths in the sample set. We also observe that developers repaired the leak defects by disposing of the unclosed resources in a `try-finally` block. This result can encourage the researcher and tool developers to improve current static analysis tools for leak detection.

<sup>50</sup><http://www.fbinfer.com>

Table 5.9: Comparison of common code transformations found in our study with previous work. 27Repairs refers to [97].

Our study	PAR	R2FIX	27Repairs
Conditional disposal of resource	✓	✓	✓
Add disposal method call	×	✓	×
Add disposal method	×	×	✓
Set obsolete reference to null	×	×	×
Add catch/try-catch block	×	×	✓
Add finally/try-finally block	×	×	×
Add try-with-resources statement	×	×	×
Change condition expression	✓	×	×
Change method call parameters	✓	×	✓
Change static object to a no-static	×	×	×
Change to weak reference	×	×	×
Replace method call	✓	×	✓
Change collection	×	×	×

In some cases, the project could not be quickly built. Hence some issues could not be detected. Note that if there is a build error, we report that issue as build error as our focus is about the applicability of Infer.

**Comparison of common code transformations.** In RQ5 (Section 5.4.5), we showed 13 common code transformation found in the studied fix patches. Previous work also reported common repair patterns [62, 74, 97]. PAR [62] found 10 manual repair patterns for Java. [74] used 8 patterns (2 of them for repairing memory leaks) to provide patches for bugs in C. [97] introduced 27 automatically extractable repair patterns.

We compare our 13 patterns with previous work to find which patterns are not reported before. Table 5.9 shows the result of our evaluation. The result shows that six code transformations were not reported before. We can also observe that “conditional disposal of resource” was also used in all studied previous work. The reason why previous work did not report some of the code transformations found by our study may be because they focused on functional bugs, while our study targets leak-related defects. We found that some of the fixes are specific for leak-related defects. For example, `try-with-resources` is only introduced to avoid potential resource leaks caused by not disposing of closable resources.

## 5.5 Implications

Based on the findings of our empirical results, we discuss the implications of our study for both researchers and practitioners.

**Prevalence of leak types.** Understanding which types of leaks are prevalent in a project can help to avoid and detect leak defects efficiently. The results of Section 5.4.1 show that each studied project has a particular dominant leak type. This knowledge can be exploited by prioritizing the most effective detection methods for the dominant leak types. As shown in Section 5.4.2, memory leaks and resource leaks have distinct best practice detection methods which can be used in a software development process. Manual code inspection is the dominant detection method for resource leaks. Projects with a large number of resource leaks can benefit from this detection method. One can further improve this by using techniques like code self-review based on the Personal Software Process (PSP) [55] with checklist items adapted for detection of resource leaks. For memory leaks, about 63% of the issues are detected or observed using the runtime information. Projects with a large number of memory leaks should consider the regular usage of the profiling tools. Profiling measures different metrics such as memory or time complexity of a program during its runtime. With this data, programmers can continuously check the resource usage of the program and react faster to the abnormal behavior.

In practical terms, the knowledge of the dominant leak types can be gained via (1) mining distribution of the leak types (or at least the dominant ones) from the bug trackers and repositories, and (2) improving the bug trackers with a labeled classification of the leaked resource.

**Good practices.** Good practices can considerably reduce the probability of introducing a leak defect. Such practices can be obtained for example from Java documentation or from existing research work. Here we describe two good practices.

*Use try-with-resources on AutoCloseable resources.* Introduced in Java 7, `try-with-resources` statement is an efficient method for better management of the closeable resources. It ensures that each resource is closed at the end of the try statement. Our empirical study shows that about 32.4% of the resource leaks are caused by shallow

exception handling. The `try-with-resources` statement can help to avoid such defects as many current Java applications support Java 7 or higher.

*Prevent having a strong reference from the value object to its key in a `HashMap`.* As opposed to regular references, weak references do not protect the objects from being disposed of by the garbage collector. This property makes them suitable for implementing cache mechanisms through *WeakHashMap*, where the entry will be disposed of as soon as the key becomes unreachable. If the value objects of a `HashMap` refers to its key, the programmer should wrap the value as *WeakReference* before putting the value into the map as recommended by the Java documentation<sup>51</sup>. Otherwise, the key cannot be discarded.

**Software testing for leak detection.** Software tests can be used as a lightweight leak detection tool. They are beneficial for decreasing the cost of leak defects by triggering the leaks before the production phase. Our study shows that over 11% of the leak defects are detected as the result of a failing test (Table 5.4). Works like [37, 41] corroborate with our results by showing the effectiveness of leak detection via testing.

**Fault localization.** Fault localization is the first step in automated program repair. Defects with high locality can be repaired with low code churn. Our results showed that leak defects are highly localized. First, in 57% of the issues, only one file was modified. This value increases to 73% for repairs with changing two files at most. Second, in about 90% of the issues, only Java files are changed. These findings can encourage researchers to improve and develop techniques for the automated repair of leak defects.

**Template-driven patch generation.** Previous works proposed patch-generation techniques based on the templates derived from existing human-written patches [62, 66]. Work [105] showed the existence of common patterns for performance problems in JavaScript. We evaluated the potential of providing template-driven repairs for leak defects through studying repair patches. We found 13 common code transformations used by developers. About 57% of the issues from patch analysis dataset are repaired by a combination of one or more of these code transformations. These results show the

---

<sup>51</sup><https://docs.oracle.com/javase/7/docs/api/java/util/WeakHashMap.html>

potential of template-driven patch generation techniques for repairing leak-inducing defects.

## 5.6 Threats to Validity

In this section, we discuss the threats to the validity of our study.

**Construct validity.** The quality of dataset used in our study is a threat to construct validity. First, we used Jira and GitHub bug tracker to collect leak-related defects. This set of defects does not necessarily include all leak defects in the studied applications. Conversely, some investigated defects might never be manifested at runtime. This might be especially the case for issues found by source-code-based detection. Second, to answer research questions, we relied on the information provided in the bug reports as they are the only source of information available. Although the bug reports in the studied projects are often high-quality reports with useful information, it is possible that the reporter provided insufficient information in the report or misdescribed the issue. However, since we investigate a large number of defects and focus on distributions and their relations, we expect that our findings describe the characteristics of the whole defect population in general.

Second, We used a simple keyword search to find leak-related bugs. Issues that do not contain the keyword “leak” can be incorrectly omitted from our data collection process. We searched for other leak-related keywords, but our query yield many false positives. To minimize such threats related to insufficient or skewed sampling of the leak defect population, we used a broad set of leak-related bugs (491 issues) from 15 large-scale projects from a variety of application categories and different software repositories.

Third, we only found one leak-related defect in our dataset in which a static analyzer detected the leak. One reason might be that the most leak-related issues are reported on a released version of an application and not during the development phase. It might be the case that the developers already used static analyzers in the development phase to remove the potential leak-related defects in the production environment.

**Internal validity.** Experimenter bias and errors are threats to internal validity. In this study, we heavily used manual analysis for categorization. When generating taxonomies defined in our study, we manually extracted the contents of the issues and used our knowledge to assign a bug to a category. To lower the risk of error in the process of manual categorization, we applied the open coding methodology. We have spent many hours studying all data related to each defect such as issue title and description, developer discussions, and repair patches.

**External validity.** Threats to external validity are related to the generalizability of our findings and implications. We collect our dataset from different categories of open source projects. The projects may not be representative for closed source projects. Our results are derived from 15 projects, and some findings may not apply to projects written in other languages. However, other managed languages share similarities with Java in terms of memory management and may benefit from some of our findings. For instance, the `open with` statement is available in Python for resource management similar to the `try-with-resources` statement in Java.

## 5.7 Chapter Summary

In this chapter, we conducted a comprehensive empirical study on 491 reported resource and memory leaks from 15 large open-source Java applications. We found that the resource leaks are slightly more often reported than memory leaks. About 45% of the resource leaks were detected using source-code analysis while more than 70% of the memory leaks were detected during runtime. We also showed that about 54% of the leak-inducing defects were repaired by modification of only one source file which implies a high localization of the repairs. Our analysis revealed that collection mismanagement and non-closed resources are the main root causes of the leak defects. We also investigated the repair patches and found 13 common code transformations for repairing the leak-inducing defects. Finally, we derived some implications from our findings which can help both researchers and practitioners for a better understanding of the characteristics of the resource and memory leaks and their repairs.



# Chapter 6

## Conclusion and Outlook

In this dissertation, we proposed approaches for automated debugging of crashing bugs and memory leak detection. We also conducted an empirical study on leak-inducing defects and their repairs to provide further insight into the properties of such defects. This section summarizes our findings and draws some directions for future work on automated debugging.

### 6.1 Conclusion

Recently software development is changed radically. With new development concepts such as agile development, each developer commits new codes to the codebase repository more than before. Therefore, the codebase of software grows in a non-linear fashion. In such a complex environment, the rate of creating new bugs after the code changes are relatively high. Therefore, the need for automated debugging and repair techniques and tools is higher than at any time.

In this thesis, we incorporated the following observations: 1) Today's software consists of many intermediate versions where two following versions differ only in a minimal subset of the code locations. 2) Each software is supported by a large number of tests to assure the software quality. Based on these observations, we tried to propose new approaches for automated debugging of functional and non-functional bugs. We highly focused on leak-inducing defects as they could profoundly affect the software quality. In addition to the newly proposed technique for leak detection

## 6. Conclusion and Outlook

based on version comparison, we conducted an empirical study to acquire a better understanding of the properties of such defects. We believe that such a study could be beneficial for both practitioners and researchers to improve the existing techniques on leak detection and diagnosis.

**Automated debugging of crashing bugs.** We have presented a scalable approach in Chapter 3 for isolation of failure-inducing changes which exploits version differences together with static and dynamic analysis. Our method had two essential strengths. First, the size of the set of suspicious statements is proportional to the size of recent code changes, making it potentially applicable to large projects. Secondly, the additional runtime overhead is on the order of executing a test triggering bug search. This allows for integrating our approach in a traditional testing process in order to enhance test outcomes with locations of potential defects.

Our preliminary evaluation of a large-scale project (Apache Hadoop) showed that results are promising, and the approach could locate some defects with high accuracy.

**Automated leak detection via regression testing.** Although only one percent of all defects in large open-source projects are memory-related issues [78], they can substantially increase the cost of ownership of large-scale software systems. Unfortunately, their inherent characteristics (namely long latency before manifestation and a weak link between defects and symptoms) substantially hamper their detection and isolation via traditional unit and integration tests.

We presented an approach for memory leak diagnosis which exploits existing tests (Chapter 4). It is based on version comparison approach in combination with data analysis. Our technique can alert about suspected presence of memory leaks and provides a ranked list of suspicious allocation sites.

Our approach showed multiple advantages. First, the effort of setup and integration into existing testing processes is low because no test modification is required and existing test processes and frameworks can be used. Therefore, this method can be integrated into the regression testing phase. In this phase, developers can apply our approach to find potential memory leaks. Secondly, the diagnosis is sufficiently accurate which can substantially shorten the repair time. Finally, the execution overhead (even in our prototypical system) is acceptable and makes it applicable in the development environment.

**Empirical study on resource and memory leaks.** Diagnosis of leak-inducing defects is one of the main challenges for both researchers and practitioners in software development and maintenance. Understanding the characteristics of resource and memory leaks can provide useful information to improve leak diagnosis techniques further. For this purpose, we conducted a detailed empirical study on a large dataset (491 issues from 15 mature projects) to understand how leaks are detected, which defects create them, and which types of repairs exist. Our findings and implications showed that even by simple changes in the quality assurance processes (e.g., code review, testing), the avoidance and diagnosis of leaks could be significantly improved.

## 6.2 Outlook

Despite the extensive research on automated debugging, we are not there yet to debug and repair all bugs automatically. Here, we suggest some directions for future research based on the results of this thesis.

The proposed approach on automated debugging of failure-inducing changes (Chapter 3) showed some limitations. First, we only used static analysis. Incorporating dynamic analysis for collecting runtime information such as the value of conditional expressions might improve the effectiveness of our approach. Second, our approach could not pinpoint the root cause of the bugs. Contrasting runtime data collected during the passing and failing run might help programmers to understand the causes of the failure.

Our approach for memory leak detection presented in chapter 4 can be improved in the following directions. First, we can set a threshold on the confidence score for automated detection of potential memory leaks. We found that such a threshold is project-specific, i.e., we need to adjust the threshold value for each project separately. We need to perform a detailed study on the impact of different threshold values on a variety of applications to find the optimal threshold value. Secondly, our approach introduced a relatively high runtime overhead. Although our approach is designed for the development and not production, we can still optimize the performance of our approach in order to reduce the runtime and memory overheads. For example, we can instrument only the part of the code which is relevant to the recent code changes.

## 6. Conclusion and Outlook

We can also execute the unit tests which only correspond to the changed code. With such optimization, one can run our approach after each commit in the development cycle. Thirdly, as our approach is highly based on the code coverage, in some cases, it resulted in a low rank for the actual leaky allocation site. A promising approach is to modify unit tests to achieve higher coverage of the modified code regions. The other direction is to generate new unit tests for triggering all possible allocation sites.

Our study in Chapter 5 provided some fruitful information about resource and memory leaks. For example, the results of our study showed that the manual runtime analysis is still the primary weapon of the practitioners for leak detection. It also showed that state-of-the-art tools (such as Infer) are not yet powerful to detect all types of resource and memory leaks. Therefore, one can ask why existing automated leak detection tools and techniques are rarely used in practice, why current tools are not helpful for the detection of all leak defects. The result of such an evaluation can pave the way for further improvement of existing leak detection tools.

From our observation, we found that there exist repair patterns for fixing leak-inducing defects. For example, we found 13 recurring code transformation. One could work on new techniques or combined the existing automated repair techniques for fixing the leak-inducing defects. One direction can be template-driven patch generation as used in previous work [62, 66].

When a new technique is proposed, it should be evaluated. One common approach is to evaluate the new technique against benchmarks. However, to our knowledge, there are no such benchmarks for leak-inducing defects. Most previous works used a limited number of leaks to evaluate their approaches. Therefore, one can implement a fault injector which simulates the distribution of the leak types and the defect types in real applications. It can serve as a practical benchmarking tool for the evaluation of methods and tools for leak diagnosis.

# Bibliography

- [1] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616, June 1993.
- [2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.
- [3] Javier Alonso, Luis Moura Silva, Artur Andrzejak, and Jordi Torres. High-available grid services through the use of virtualized clustering. In *IEEE-GRID*, Austin, USA, September 2007.
- [4] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d’Amorim. Fault-localization using dynamic slicing and change impact analysis. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 520–523, Washington, DC, USA, 2011. IEEE Computer Society.
- [5] Amazon AWS. Summary of the October 22,2012 AWS Service Event in the US-East Region. <https://aws.amazon.com/de/message/680342>, 2012.
- [6] Artur Andrzejak and Luis Silva. Deterministic models of software aging and optimal rejuvenation schedules. In *10th IFIP/IEEE Symposium on Integrated Management (IM 2007)*, Munich, Germany, May 2007.
- [7] David Andrzejewski, Anne Mulhern, Ben Liblit, and Xiaojin Zhu. Statistical debugging using latent topic models. In *Proceedings of the 18th European Conference on Machine Learning*, ECML '07, pages 6–17, Berlin, Heidelberg, 2007. Springer-Verlag.
- [8] J. Araujo, R. Matos, P. Maciel, and R. Matias. Software aging issues on the eucalyptus cloud computing infrastructure. In *Proc. IEEE Int Systems, Man, and Cybernetics (SMC) Conf*, pages 1411–1416, 2011.
- [9] Ron Artstein and Massimo Poesio. Inter-coder agreement for computational linguistics. *Comput. Linguist.*, 34(4):555–596, December 2008.

- [10] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07*, pages 5–15, New York, NY, USA, 2007. ACM.
- [11] Piramanayagam Arumuga Nainar and Ben Liblit. Adaptive bug isolation. In *ICSE*, 2010.
- [12] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering*, 44(5):470–490, May 2018.
- [13] Ansuman Banerjee, Abhik Roychoudhury, Johannes A. Harlie, and Zhenkai Liang. Golden implementation driven software debugging. In *FSE*, pages 177–186, 2010.
- [14] Michael D. Bond and Kathryn S. McKinley. Bell: Bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII*, pages 61–72, 2006.
- [15] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA '08*, pages 109–126, 2008.
- [16] Michael D. Bond and Kathryn S. McKinley. Leak Pruning. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–288, 2009.
- [17] Michael D. Bond and Kathryn S. McKinley. Leak pruning. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS XIV, pages 277–288, 2009.
- [18] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with tarantula. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability, ISSRE '07*, pages 137–146, Washington, DC, USA, 2007. IEEE Computer Society.
- [19] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering, ICSE '04*, pages 480–490, Washington, DC, USA, 2004. IEEE Computer Society.
- [20] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS*, pages 125–130. IEEE Computer Society, 2001.

- [21] V. Castelli, R. Harper, P. Heidelberg, S. Hunter, K. Trivedi, K. Vaidyanathan, and W. Zeggert. Proactive management of software aging. *IBM Journal Research & Development*, 45(2), March 2001.
- [22] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Formal concept analysis enhances fault localization in software. In *Proceedings of the 6th International Conference on Formal Concept Analysis*, ICFCA’08, pages 273–288, 2008.
- [23] Kung Chen and Ju-Bing Chen. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *IEEE International Conference on Computers, Software & Applications (COMPSAC)*, pages 23–28, 2007.
- [24] Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An empirical study on the practice of maintaining object-relational mapping code in java systems. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR ’16, pages 165–176, 2016.
- [25] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical Memory Leak Detection Using Guarded Value-flow Analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 480–491, 2007.
- [26] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM ’04, pages 85–96, 2004.
- [27] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, pages 34–44, 2009.
- [28] James Clause and Alessandro Orso. LEAKPOINT: Pinpointing the Causes of Memory Leaks. In *International Conference on Software Engineering (ICSE)*, pages 515–524, 2010.
- [29] James Clause and Alessandro Orso. Leakpoint: Pinpointing the causes of memory leaks. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE ’10, pages 515–524, 2010.
- [30] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.
- [31] James S. Collofello and Scott N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *J. Syst. Softw.*, 9(3):191–195, March 1989.
- [32] Richard Demillo, Hsin Pan, and Eugene Spafford. Failure and fault analysis for software debugging. In *Proceedings - IEEE Computer Society’s International Computer Software and Applications Conference*, pages 515 – 521, 09 1997.

- [33] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '96, pages 121–134, 1996.
- [34] Isil Dillig, Thomas Dillig, Eran Yahav, and Satish Chandra. The closer: Automating resource management in java. In *ACM International Symposium on Memory Management (ISMM)*, pages 1–10, 2008.
- [35] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw.*, 83(2):188–208, February 2010.
- [36] W. Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19, 11 2011.
- [37] Lu Fang, Liang Dou, and Guoqing (Harry) Xu. Perfblower: Quickly detecting memory-related performance problems via amplification. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 296–320, 2015.
- [38] The Eclipse Foundation. The Eclipse Memory Analyzer (MAT). Version 1.2.
- [39] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 459–470, 2015.
- [40] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. Trivedi. A methodology for detection and estimation of software aging. In *Proceedings of the 9th Int'l Symposium on Software Reliability Engineering*, pages 282–292, 1998.
- [41] Mohammadreza Ghanavati and Artur Andrzejak. Automated memory leak diagnosis by regression testing. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*, pages 191–200, 2015.
- [42] Mohammadreza Ghanavati, Diego Costa, Artur Andrzejak, and Janos Seboek. Memory and resource leak defects in java projects: An empirical study. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '18 Companion, pages 410–411, 2018.
- [43] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. Memory and resource leak defects and their repairs in java projects. *CoRR*, abs/1810.00101, 2018.
- [44] Ross Gore, Paul F. Reynolds, and David Kamensky. Statistical debugging with elastic predicates. In *Proceedings of the 2011 26th IEEE/ACM International*



- Conference on Automated Software Engineering*, ASE '11, pages 492–495, Washington, DC, USA, 2011. IEEE Computer Society.
- [45] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. Characterizing and detecting resource leaks in android applications. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 389–398, 2013.
  - [46] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE*, pages 263–272, 2005.
  - [47] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.
  - [48] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88. IEEE Computer Society, 2009.
  - [49] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX*, pages 125–138, 1991.
  - [50] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 156–164, 2004.
  - [51] David L. Heine and Monica S. Lam. Static Detection of Leaks in Polymorphic Containers. In *International Conference on Software Engineering (ICSE)*, pages 252–261, 2006.
  - [52] Susan. Horwitz, Thomas. Reps, and David. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.
  - [53] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of Fault-Tolerant Computing Symposium FTCS-25*, June 1995.
  - [54] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management*. John Wiley & Sons, USA, 2007.
  - [55] Watts Humphrey. The personal software process (psp). Technical Report CMU/SEI-2000-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000.
  - [56] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, pages 2–10, 1994.

- [57] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, 2012.
- [58] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.
- [59] Maria Jump and Kathryn S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 31–38, 2007.
- [60] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. Automated Memory Leak Detection for Production Use. In *International Conference on Software Engineering (ICSE)*, pages 825–836, 2014.
- [61] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 825–836, 2014.
- [62] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.
- [63] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, October 1988.
- [64] Felix Langner and Artur Andrzejak. Detecting software aging in a cloud computing framework by comparing development versions. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 896–899, 2013.
- [65] Felix Langner and Artur Andrzejak. Detection and root cause analysis of memory-related software aging defects by automated tests. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 365–369, 2013.
- [66] Xuan-Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering*, SANER '16. IEEE, 2016.
- [67] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, September 1980.
- [68] L. Li, K. Vaidyanathan, and K. Trivedi. An approach for estimation of software aging in a web-server. In *ISESE'02*, pages 91–102, 2002.

- [69] Ben Liblit. *Cooperative Bug Isolation: Winning Thesis of the 2005 ACM Doctoral Dissertation Competition*, volume 4440 of *Lecture Notes in Computer Science*. Springer, 2007.
- [70] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.
- [71] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.
- [72] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, October 2006.
- [73] Chao Liu, Xiangyu Zhang, Yu Zhang, Jiawei Han, and Bharat K. Bhargava. Indexing noncrashing failures: A dynamic program slicing-based approach. In *International Conference Software Maintenance (ICSM)*, pages 455–464. IEEE, 2007.
- [74] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. R2fix: Automatically generating bug fixes from bug reports. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, ICST '13*, pages 282–291, 2013.
- [75] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. How practitioners perceive the relevance of software engineering research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 415–425, 2015.
- [76] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16*, pages 298–312, 2016.
- [77] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *International Conference on Computers and Applications*, pages 877–882, 1987.
- [78] Fumio Machida, Jianwen Xiang, Kumiko Tadano, and Yoshiharu Maeno. Aging-related bugs in cloud computing software. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops, ISSREW '12*, pages 287–292, 2012.
- [79] Jeremy Manson. java-allocation-instrumenter: A Java agent that rewrites bytecode to instrument allocation sites. <https://code.google.com/p/java-allocation-instrumenter/>, February 2012.
- [80] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling java programs for diagnosis. In *Proceedings of the 14th European Conference on*

- Artificial Intelligence*, ECAI'00, pages 171–175, Amsterdam, The Netherlands, The Netherlands, 2000. IOS Press.
- [81] R. Matias, A. Andrzejak, F. Machida, D. Elias, and K. Trivedi. A systematic differential analysis for fast and robust detection of software aging. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 311–320, Oct 2014.
  - [82] Rivalino Matias, Artur Andrzejak, Fumio Machida, Diego Elias, and Kishor Trivedi. A systematic approach for low-latency and robust detection of software aging. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2014.
  - [83] Steve McConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, USA, 1993.
  - [84] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, 2016.
  - [85] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.
  - [86] Bryan Meredith. Omega - An Instant Leak Detection Tool for Valgrind, 2008. Version 1.2.
  - [87] Ghassan Mishserghi and Zhendong Su. Hdd: hierarchical delta debugging. In *ICSE*, pages 142–151, 2006.
  - [88] Nick Mitchell and Gary Sevitsky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 351–377, 2003.
  - [89] Syeda Nessa, Muhammad Abedin, W. Eric Wong, Latifur Khan, and Yu Qi. Software fault localization using n-gram analysis. In *Proceedings of the Third International Conference on Wireless Algorithms, Systems, and Applications*, WASA '08, pages 548–559, 2008.
  - [90] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.
  - [91] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of*

- the 2013 International Conference on Software Engineering, ICSE '13*, pages 772–781, 2013.
- [92] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13*, pages 237–246, 2013.
  - [93] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and Precisely Locating Memory Leaks and Bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 397–407, 2009.
  - [94] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, pages 397–407, 2009.
  - [95] Maksim Orlovich and Radu Rugina. Memory Leak Analysis by Contradiction. In *International Static Analysis Symposium (SAS)*, pages 405–424, 2006.
  - [96] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000–2014). In *Conference on Future of Software Engineering (FOSE)*, pages 117–132, 2014.
  - [97] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.
  - [98] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011.
  - [99] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering, ICSE '17*, pages 609–620, Piscataway, NJ, USA, 2017. IEEE Press.
  - [100] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/FSE*, pages 33–42, 2009.
  - [101] Derek Rayside and Lucy Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, pages 194–203, 2007.
  - [102] Jeremias Roessler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In *ISSTA*, pages 309–319, 2012.

- [103] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.*, 25(4):557–572, July 1999.
- [104] Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. Defect categorization: Making use of a decade of widely varying historical data. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 149–157, 2008.
- [105] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: An empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 61–72, 2016.
- [106] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Mooly Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 483–503, 2003.
- [107] Luis Moura Silva, Javier Alonso, Paulo Silva, Jordi Torres, and Artur Andrzejak. Using virtualization to improve software rejuvenation. In *IEEE International Symposium on Network Computing and Applications (IEEE-NCA)*, Cambridge, MA, USA, July 2007.
- [108] Luis Moura Silva, Henrique Madeira, and João Gabriel Silva. Software aging and rejuvenation in a SOAP-based server. In *IEEE International Symposium on Network Computing and Applications (IEEE-NCA)*, pages 56–65, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
- [109] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010.
- [110] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 561–578, 2014.
- [111] V. Sor, P. Oü, T. Treier, and S. N. Srirama. Improving statistical approach for memory leak detection using machine learning. In *2013 IEEE International Conference on Software Maintenance*, pages 544–547, Sept 2013.
- [112] Vladimir Šor. *Statistical approach for memory leak detection in Java applications*. PhD thesis, University of Tartu, Estonia, Tartu, Estonia, 2014.
- [113] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *PLDI*, pages 112–122, 2007.
- [114] Chad D. Sterling and Ronald A. Olsson. Automated bug isolation via program chipping. *Software: Practice and Experience*, 37(10):1061–1086, 2007.

- [115] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Softw. Engg.*, 19(6):1665–1705, December 2014.
- [116] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 535–544, 2010.
- [117] K. Vaidyanathan and K. S. Trivedi. A measurement-based model for estimation of resource exhaustion in operational software systems. In *Proceedings of 10th IEEE Int'l Symposium on Software Reliability Engineering*, pages 84–93, November 1999.
- [118] Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 151–162, 2018.
- [119] WALA. <http://sourceforge.net/projects/wala/>.
- [120] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtii. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 98:98–98:108, 2014.
- [121] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 419–431, 2004.
- [122] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, 2005.
- [123] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, 2009.
- [124] Mark Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, 1984.
- [125] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Softw. Eng.*, 42(8):707–740, August 2016.
- [126] Franz Wotawa. On the relationship between model-based debugging and program slicing. *Artif. Intell.*, 135(1-2):125–143, February 2002.

- [127] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In *Proceedings of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Developments in Applied Artificial Intelligence*, pages 746–757, 2002.
- [128] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: Recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE ’11, pages 15–25, 2011.
- [129] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. Leakchaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’11, pages 270–282, 2011.
- [130] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FSE/SDP Workshop on Future of Software Engineering Research (FoSER)*, pages 421–426, 2010.
- [131] Guoqing Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE ’08, pages 151–160, 2008.
- [132] Guoqing Xu and Atanas Rountev. Precise Memory Leak Detection for Java Software Using Container Profiling. *ACM Transactions on Software Engineering and Methodology*, 22(3):17:1–17:28, July 2013.
- [133] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. Leakchecker: Practical static memory leak detection for managed languages. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO ’14, pages 87:87–87:97, 2014.
- [134] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Automated memory leak fixing on value-flow slices for c programs. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC ’16, pages 1386–1393, 2016.
- [135] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE ’11, pages 26–36, 2011.
- [136] S Yoo, X Xie, Fei-Ching Kuo, T.Y Chen, and Mark Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. *Technical Report RN/14/14*, 01 2014.



- [137] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. Practical isolation of failure-inducing changes for debugging regression faults. In *ASE*, ASE 2012, pages 20–29, 2012.
- [138] Andreas Zeller. Yesterday, My Program Worked. Today, It Does Not. Why? In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 253–267, 1999.
- [139] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, 1999.
- [140] Andreas Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, 2002.
- [141] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
- [142] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *PLDI*, pages 169–180, 2006.
- [143] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faulty code by multiple points slicing. *Software: Practice and Experience*, 37(9):935–961, 2007.
- [144] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Softw. Engg.*, 12(2):143–160, April 2007.
- [145] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *ICSE*, pages 319–329, 2003.
- [146] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 502–511, Washington, DC, USA, 2004. IEEE Computer Society.
- [147] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 913–923, 2015.