Study and Prioritization of Dependency-Oriented Test Code for Efficient Regression Testing

by

SHAIKH MOSTAFA, B. Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

COMMITTEE MEMBERS:
Xiaoyin Wang, Ph.D., Chair
Jianwei Niu, Ph.D.
Wei Wang, Ph.D.
Palden Lama, Ph.D.
Lide Duan, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
March 2018

DEDICATION

I would like to dedicate this thesis/dissertation to my parents.

ACKNOWLEDGEMENTS

First of all, I would like to thank my adviser, Xiaoyin Wang. If it were not for him, I would neither have started nor finished my PhD. I very much enjoyed spending time and working with him. I have to thank him for numerous nights of assisting on the project or conversation through email. His advises regarding my research as well as professional career have been invaluable. Although I could never do enough to return all that he has done for me, I hope to assist the junior co-workers with the same passion in the future.

I would like to convey my deepest gratitude to Tao Xie for being a tremendous mentor of the research project PerfRanker. Furthermore, I would like to thank Jianwei Niu, Palden Lama, Wei Wang, and Lide Duan for their generous help during my graduate studies. They served on my thesis committee and helped me improve presentation of this material.

My internship mentors also provided great support and lovely environments: John Micco (Google) and Abhayendra Shing (Google) to carry out a research project on google infrastructure. I was honored to work with exceptional master's student Rodney Rodriguez. I would like to thank other colleagues and friends, including Foyzul Hasan and Xue Qin.

Last but not least, I would like to thank my sister, my brother, my parents and my wife for their never-ending love, care, and support.

This Doctoral Dissertation Thesis was produced in accordance with guidelines which permit the inclusion as part of the Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgement to be made of the importance and originality of the research reported.

It is acceptable for this Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in Doctoral Dissertation attest to the accuracy of this statement.

March 2018

Study and Prioritization of Dependency-Oriented Test Code for Efficient Regression Testing

Shaikh Mostafa, Ph. D.
The University of Texas at San Antonio, 2018

Supervising Professors: Xiaoyin Wang, Ph.D.

In modern software process, software testing is performed along with software development to detect errors as early as possible and to guarantee that changes made in software do not affect the system negatively. However, during the development phase, the test suite is frequently expanded for new features and tends to increase in size quickly. Due to the resource and time constraints for re-executing large test suites, it is important to develop techniques to reduce the effort of regression testing.

Unit testing testing is the core of test driven development where software testers need to test a class or a component without integration with some of its dependencies. Typical reasons for excluding dependencies in testing include high cost of invoking some dependencies (e.g., slow network or database operations, commercial third-party web services), and the potential interference of bugs in dependencies. In practice, mock objects have been used in software testing to simulate such missing dependencies. However, due to exclusion of dependencies, mock-objects-based testing is not suitable for performance regression and backward incompatibility regression testing.

A small extent of performance degradation may result in severe consequence and running performance test needs more time and resources. Also, it can be hard for a developer to understand performance impact with few runs. Furthermore, A code changes touches several test cases is very common during the evolution of software. Due to the resource and time constraints for re-executing large test suites, it is important to develop techniques to reduce the effort of regression

testing. Our proposed method focus on the performance test suite prioritization via performance impact analysis of change.

Nowadays, due to the frequent technological innovation and market changes, software libraries are evolving very quickly. To make sure that existing client software applications are not broken after a library update, backward compatibility has always been one of the most important requirements during the evolution of software platforms and libraries. Previous studies on this topic mainly focus on API signature changes between consecutive versions of software libraries, but behavioral changes of APIs with untouched signatures are actually more dangerous and are causing most real world bugs because they cannot be easily detected. Our study categorizes behavioral backward incompatibilities according to incompatible behaviors and invocation conditions. We propose to compare those detected in regression testing with those causing real-world bugs and prioritizing test case based on backward incompatibilities in dependencies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: INTRODUCTION

1.1 Motivation

In every aspect of our lives, e.g., ranging from communication to social networks to entertainment to business to transportation to health uses of software is predominant. Therefore, software correctness is of utmost importance. The world has witnessed the high cost of bugs far too many times. Prior studies in the area of software testing estimate that bugs cost global economy more than $300 billion per year. Despite the risk of introducing new bugs while making changes, software constantly evolves due to never-ending requirements. When Agile software development models were first envisioned, a core tenet was to iterate more quickly on software changes and determine the correct path via exploration–essentially, striving to "fail fast" and iterate to correctness as a fundamental project goal.

Thus, software developers have to check, at each project revision, not only correctness of newly added functionality, but also that the recent project changes did not break any previously working functionality. Software testing is the most common approach in industry to check correctness of software. Software developers usually write tests for newly implemented functionality and include these tests in a test suite (i.e., a set of tests for the entire project). To check that project changes do not break previously working functionality, developers practice regression testing - running test suite at each project revision. Clearly, more automation and better tools for software testing can lower the cost of software development, increase the reliability of software, and reduce the negative economic impact of defective software. So, continuous integration process which automatically compile, build, and test every new version of code committed to the central team repository, ensures that the entire team is alerted any time the central code repository contains broken code.

Although regression testing is important, it is costly because it frequently runs a large number of tests. Some research studies [93] [28] [38] [72] estimate that regression testing can take up to

80% of the testing budget and up to 50% of the software maintenance cost. The cost of regression testing increases as software grows. For example, Google reported that In 2014, approximately 15 million lines of code were changed in approximately 250,000 files in the Google repository on a weekly basis. Google's code base is shared by more than 25,000 Google software developers from dozens of offices in countries around the world. On a typical workday, they commit 16,000 changes to the code base, and another 24,000 changes are committed by automated systems. Their regression-testing system, TAP [5] [6] [35], has had a linear increase in both the number of project changes per day and the average test-suite execution time per change, leading to a quadratic increase in the total test-suite execution time per day. As a result, the increase is challenging to keep up with even for a company with an abundance of computing resources. Other companies and open-source projects also reported long regression testing time.

Due to the resource and time constraints for re-executing large test suites, it is important to develop techniques to reduce the effort of regression testing. Unit testing testing is the core of test driven development where software testers need to test a class or a component without integration with some of its dependencies. Typical reasons for excluding dependencies in testing high cost of invoking some dependencies (e.g., slow network or database operations, commercial third-party web services), and the potential interference of bugs in the dependencies. In practice, mock objects have been used in software testing to simulate such missing dependencies. However, due to exclusion of dependencies, mock-objects-based testing is not suitable for performance regression and backward incompatibility regression testing.

A small extent of performance degradation may result in severe consequence motivate us that running performance test needs more time and resources. Also, it can be hard for a developer to understand performance impact with few runs. Furthermore, A code changes touches several test cases is very common during the evolution of software. Due to the resource and time constraints for re-executing large test suites, it is important to develop techniques to reduce the effort

of regression testing. Our proposed method focus on the performance test suite prioritization via performance impact analysis of change.

Nowadays, due to the frequent technological innovation and market changes, software libraries are evolving very quickly. To make sure that existing client software applications are not broken after a library update, backward compatibility has always been one of the most important requirements during the evolution of software platforms and libraries. Previous studies on this topic mainly focus on API signature changes between consecutive versions of software libraries, but behavioral changes of APIs with untouched signatures are actually more dangerous and are causing most real world bugs because they cannot be easily detected. Our study categorizes behavioral backward incompatibilities according to incompatible behaviors and invocation conditions. We compared those detected in regression testing with those causing real-world bugs using our defined category and presented the resolution of of real bugs related to behavioral backward incompatibilities.

1.2 Thesis Statement

Our thesis is three-pronged:

> ***(1)*** *Developer should have better understanding when mock and when real object*
>
> ***(2)*** *There is a need for an automated performance test suite prioritization technique for collection-intensive software that works in practice*
>
> ***(3)*** *It is important to study and categorization of behavioral backward incompatibilities*

Our vision is to provide a framework to determine when to do mocking in testing and when to use the real dependency. A more precise question should be, developers should do mocking for which test cases and include all dependencies for which test cases.

In the figure 1.1, we can see the smart mocking system takes a code commit, the software's

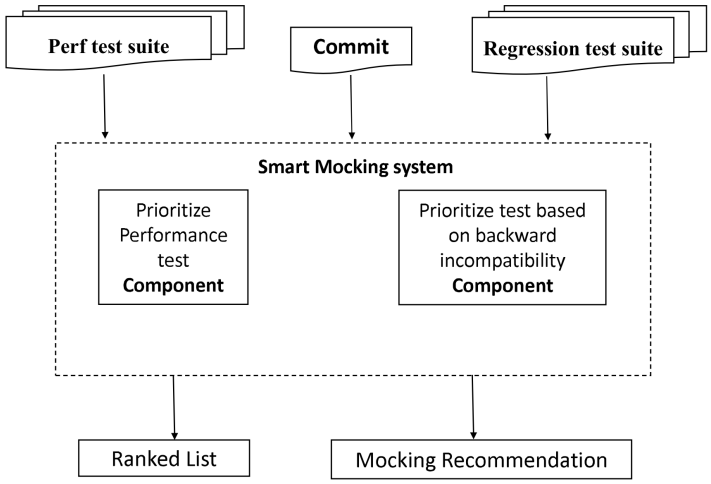

Figure 1.1: Smart Mocking System

performance test suite, and unit test suite as the input, and its output is a list of test cases that should not be mocked based on performance impact and behavioral incompatibility. There are two major component of the system, one is performance impact analysis component and the other is behavioral incompatibility analysis component. It should be noted that these two component can not only provide recommendations on which test cases to mock and for which test cases we should replace mock by real code, but also rank the test cases on their importance.

1.3 Contributions

To confirm the thesis statement, this dissertation makes the following contributions:

- The dissertation presents the first empirical study on status and the usage of mocking frameworks in software testing. The study shows that mocking frameworks are widely used in practice, and a small portion of dependencies are mocked. Software testers tend to mock source code classes than libraries, while library classes also take a substantial proportion (40%) in all mocked classes and developer most likely to mock network, database and time consuming services API.

- The main goal of this dissertation project is to investigate and prioritize software regression

testing and how we can improve the efficiency and cost reduction of software regression testing. Particular, given a large number of existing performance test cases, how should we rank them such that commit of code changes are pushed to repository in a given time. The dissertation introduces a novel technique for performance RTS, named PerfRanker and our evaluation results show that, compared with the best of the three other baseline approaches, our approach achieves an average improvement of 17.6 percentage points on APFD-P and 27.4 percentage points on DCG. Furthermore, for Apache Commons Math and Xalan, our approach is able to rank top 1 affected test case within top 8% and top 16% test cases, and top 3 affected test cases within top 37% and 30% test cases, respectively.

- The dissertation presents study and categorization behavioral backward incompatibilities according to incompatible behaviors and invocation conditions beyond api signature. The study shows that behavioral backward incompatibilities are prevalent among Java software libraries, and caused most of real-world backward-incompatibility bugs. Furthermore, many of the behavioral backward incompatibilities are intentional, but are rarely well documented.

1.4 Organization

This dissertation is organized as follows. Chapter 2 introduces the background and related work. Chapter 3 describes our empirical study on the usage of mocking frameworks in software testing. Chapter 4 presents performance test case prioritization and experimental results for them. Chapter 5 describe our study categorization of behavioral backward incompatibilities according to incompatible behaviors and invocation conditions. In chapter 6 and 7 we discusses the lesson learned and future work direction.

Chapter 2: BACKGROUND AND RELATED WORK

The purpose of this section is to provide the background of this study and a review of related literature. First, the background is introduced followed by a related work section about Regression Testing, Mocking, Performance Regression Testing and Backward Incompatibility.

2.1 Regression Testing

Regression testing is the activity of testing software after it has been modified to gain confidence that the newly introduced changes do not obstruct the behavior of the existing, unchanged parts of the software. In order to gain confidence that program changes did not introduce any errors, regression test suites are executed recurrently. There are different variant of regression test such as unit level, integration level, functional level and system level. In each of them number of test cases can greatly influence the execution time of a test suite. Regression testing can be either manual or automated. Manual regression testing is performed by manual testing engineers. In this case, regression testing does not require any testing tools except those for bug reporting. Automated testing is performed when a code change committed to the source code repository that trigger all the existing test cases automatically. There are a number of challenges related to regression testing, such as identification of obsolete test cases, regression test selection, prioritization and minimization and test suite augmentation [131]. Yoo and Harman [133] conducted a survey on regression testing minimization, selection and prioritization, constituting nearly 200 papers. It encompasses the main research results around regression testing, addressing the problems of identifying obsolete, reusable and re-testable test cases (selection), eliminating redundant test cases (minimization) and ordering test cases to maximize early fault detection (prioritization).

2.2 Mocking

Programmer most often write unit tests - small scope (e.g. a class, a function or a component) and one of the key characteristics of unit testing is fast - mean that they can be run very frequently.If

the programmer introduced the defect with the last change, it is much easier for the person to spot the bug because does not have to look far. But what happen when the method depends on system-ddl, network, database or servlet engine? Because running those method introduce more cost to regression testing. In practice those dependency are actually replaced my mocking object which is a fake object or proxy of a real object.

Mocking objects are used in software testing to simulate software dependencies so that the testing process can be accelerated and the testing scope can be limited to the component under test (instead of going beyond the interface of dependencies and invoke potential bugs relevant to dependencies). To simulate real dependencies, mock objects typically have the same interface as the objects they mimic. Therefore, the client object remains unaware of whether it is using a real object or a mock object.

For example, when testing an email client, software testers may not use the real email server which has slow response and random failures. By contrast, they may construct a fake email server object (a mock object) that returns whatever expected by the software testers. The expected return value can be either a simple success code for the testing of normal functioning of the email client, or a server error code for the testing of error handling of the email client. Such a fake email server helps to reduce testing time as well as to explore the handling of server errors that are very hard to invoke with a real server. However, when such a simplified email server is used, it is hard for testers to tell whether the email client has invoked the email server in a correct order (if multiple-step configuration of the server is required), or whether the client has passed correct arguments to the server.

Therefore, the software testers may further have some code in the fake email server to check whether the client interacts with it correctly and passes correct arguments. With such a mock email server, software testers can perform quick and controllable testing of the email client without losing the thorough exploration of the interface between the email client and the email server.

2.3 Mocking Frameworks

As mentioned in the previous section, mocking objects are desirable for simulating software dependencies in software testing. However, since mock objects need to perform various checking of the invocations, it is usually non trivial to write such mock objects, and thus the benefit of using mock objects can be undermined by the effort spent on writing mock objects.

To solve this problem, in the past decade, people have developed various mocking frameworks to generate mock objects automatically. A code sample of using a mocking framework is presented below. With a mocking framework, software testers first create an empty mock object (Line 3). Then, for the created mock object, software testers can specify a number of invocation for the object to check, both on the order of the invocations and the correctness of the arguments (Line 6). After that, in the test code, testers run the component under test as usual but with created mock objects (Line 8). During the execution, if the specified invocation order or arguments are not satisfied, the mock object will throw a test failure so that the software testers know that something wrong happened (Line 9).

```
1: @Test
2: public void testEmailClient(){
3:      Server sv = EasyMock.
            CreateMock(Server.class);
4:      Client c = new Client(sv);
5:      Email em = new Email(...)
         //record
6:      EasyMock.expect(sv.send(em)).
            andReturn(0);
         //replay
7:      EasyMock.replay(sv);
8:      c.send(em)
9:      EasyMock.verify(sv)
10:}
```

For each major programming language, there have been a number of mocking frameworks developed, such as Mockito, EasyMock, and JMock for Java, pmock, mox for Python, and NMock, Moq for C#. All of these mocking frameworks support the basic features of mocking including the

creation of mock objects, the specification of invocation orders and arguments, and the verification of the specifications. The differences between these mocking frameworks are mainly on the design of APIs and supporting of advanced specifications such as checking the number of invocations of a certain method.

There have been a number of existing research efforts on enhancing mocking frameworks or leveraging mock objects to improve automatic software testing. Freeman et al. [45] [44] presented the basic process and concepts of using mocks objects in unit testing, as well as a mocking framework jMock. Galler et al. [46] proposed an approach to automatically generation of mocking objects satisfying certain preconditions to serve as test inputs in automatic test-case generation for Java unit testing. Taneja et al. [117] proposed to automatically generate mock objects to simulate the behavior of database systems in the automatic test-case generation of database systems. Coelho et al. [29] proposed an approach to generate mock agents in the unit testing of multi-agent software systems. Due to the necessity of mock objects in unit testing, researchers also proposed approaches to automatically generating mock objects along with unit test cases [57] [99]. For the existing test cases which are not using mock objects, Saff and Ernst [112] proposed an approach to automatically refactor such test cases by adding mock objects. Marri et al. [81] carried out an experiment to study the benefit of using mock objects to simulate file systems. All these research efforts are about automatic generation of mock objects and how to leverage mock objects in specific testing problems, while our work focuses on the current usage status of mocking frameworks and mock objects in real world software projects. As far as we know, this paper is the first research effort to study the usage status of mocking frameworks and mock objects in software practice.

Finally, the code becoming more isolated as mocking replacing real dependency and missing performance and behavioral impact of a code change. So there is need of understanding when mock and when real code.

2.4 Performance Regression Testing

Software change is inevitable as business requirement changes added new code, as well as modifications to pre-existing code to enhance and fix existing software. These continuous software changes may introduce bug or affect software runtime performance. It is often perceived that the longer it takes to discover performance loss, the harder it is to remove the business loss. Performance regression testing is an effective way to identify performance regressions in early stages. Performance test cases often involves larger inputs and thus consumes more resources and take more time. For example, performance test suite of Web Server takes 3 min to 1 hr, Database takes 10 min to 3 hrs, Compiler takes 1 hr to 20 hrs and OS takes 2 hrs to 24 hrs.Theoretically, for each test case that does not involve random process (e.g., generating random numbers, randomly selecting next method invocations), all of its executions should go over exactly the same instruction sequence, and thus take exactly the same time. However, in reality, modern mechanisms in hardware and software may bring in various random factors. Some well known examples are the randomness in scheduling of cores and buses in multi-core systems, in caching policies, and in the garbage collection process, etc. These factors interact with each other and may further amplify their effect so that the execution time of a test case may change a lot from time to time. To neutralize such randomness, most researchers and testers execute a test case multiple times and calculate the average performance and performance regression testing is expensive to conduct frequently. Developers can apply systematic, continuous performance regression testing to reveal such performance regressions in early stages. The performance variance among different executions of a same test case and a code change may touches large number of test cases provides us an even stronger motivation for test prioritization in performance regression testing.

The default approach for regression testing is to retest all test cases after releasing a new version, which is an expensive proposition. To solve this problem, there are good collection of industry case studies and research effort on performance regression testing in software systems. For example, automating regression benchmarking [61], a model-based performance testing framework for

workloads [15], genetic algorithm to expose performance regression [79], learning-based performance testing framework for test input data [49], symbolic execution to generate load test [137] and probabilistic symbolic execution [26] focus on building better performance regression testing infrastructure and test cases. Other important works are performance bug detection([59,60,62,132]), performance debugging([10,53,71,114]), automatic fixing performance problem [94] and performance regression testing result analysis([40,41]) focus on detecting performance regression bugs or provide forensic diagnosis. But Our work assumes the existence performance testing infrastructure and test cases, and improves the testing efficiency by prioritizing test suite.

Functional regression is a well research area to reduce testing cost by test case selection based on test case property and code modification([27, 110]), test suite reduction by removing redundancy in test suite ([20, 55, 138]) and test cases prioritization orders test case execution in a way to hope to detect functional fault faster([36, 63, 74, 111]). Different from these work, our goal is to reduce performance regression testing overhead via test suite prioritization based on change impact analysis whether an operation is expensive or lies in hot path. Most relevant work on the performance risk implication of code change [56]. However, this work relies on static analysis and focuses on specific types of performance regressions. Furthermore, prioritizing commits is not enough to address performance regression because a code changes touches several test cases is very common during the evolution of software. In our study, we found that some commits touches more than 100 test cases.

2.4.1 Test Prioritization

Test-case prioritization is a well studied research area. As for generic prioritization strategies, the total and additional strategies are the most widely-used prioritization strategies [111], and reported empirical results show that the additional strategy is more effective than the total strategy in most cases. There also have been a number of research efforts seeking for other optimal prioritization strategies. For example, Li et al. [74] proposed a 2-optimal strategy based on two different

strategies: hill-climbing, and genetic programming. respectively. Jiang et al. [58] proposed an adaptive random strategy for test-case prioritization. Bryce and Memon [24], which aims to prioritize test cases (i.e., event sequences) for event-based GUI software. As each test case is an event sequence in GUI testing, their approach tries to select event sequences to cover different event interactions as early as possible. Zhang et al. [136] proposed a generic strategy that has flavor of both total and additional strategies.

Besides the generic prioritization strategy, a criteria is necessary for test-case prioritization for comparing test cases. Most existing efforts in this area leverage as criteria the code coverage at different levels. There have been work based on statement and branch coverage [111], function coverage [37], block coverage [34], modified condition/decision coverage [34], etc. There have also been research [66] on test-case prioritization using coverage of system models. Mei et al. [86] investigate criteria based on dataflow coverage for testing service-oriented software.

Compared with existing techniques on prioritization of test executions, the work of this thesis focuses on the prioritization of the tests to be not mocked, so it mainly focuses on the influence of a code change on software performance and library dependencies.

2.5 Backward Incompatibility

Nowadays, as software products become larger and more complicated, software libraries have become a necessary part of almost any software. Since software libraries and their client software are typically maintained by different developers, the asynchronous evolution of software libraries and client software may result in incompatibilities. To avoid incompatibilities, for decades, "backward compatibility" has been a well known requirement in the evolution of software libraries. Each API method in an existing version of software library should exactly maintain its behavior in the following versions. API changes can be classified into api signature change and api signature is untouched but behavior of the api is change. The former is easy to detect during compile time but the later is more difficult to detect and may have severe consequence on the success of the software.

Researchers have noticed that software libraries are evolving frequently for a long time, so a number of studies have been conducted on the evolution of software libraries. Raemaekers et al. [106] proposed a measurement of software-library stability which considers API method difference and code difference, and studied the stability of 140 industrial Java systems based on the measurement. McDonnell et al. [85] studied the stability of Android APIs (in terms of added and removed classes and methods), and the time lag between the release of API changes and the corresponding adaptation at the client software side. Espinha et al. [39] interviewed 6 web client software developers and conducted an empirical study on four widely used web services to understand their API evolution trends, including the frequency of API changes, and the time given client developers to upgrade to the new version of services. Bavota et al. [16, 17], studied the evolution of software dependency upgrades in the apache software ecosystem. Robbes et al. [109] studied the reaction of developers to deprecated APIs in SmallTalk ecosystem. The existing research efforts mainly focus on signature-level API changes (Raemaekers et al.'s work further considers the amount of code difference) to measure API changes and stability. By contrast, our study focuses on behavioral changes of software libraries, which are more difficult to be detected and may cause more severe consequences.

Chapter 3: AN EMPIRICAL STUDY ON THE USAGE OF MOCKING FRAMEWORKS IN SOFTWARE TESTING

3.1 Introduction

In modern software development, testing is a major approach to software quality assurance before a software application is released. Though the main purpose of testing is to reveal errors in the software application itself, in modern software practice, software applications can hardly be executed and tested without any dependencies such as libraries, third-party services, external data, etc.

To perform testing efficiently and effectively, one significant challenge is to handle various software dependencies properly. It should be noted that testing a software application with all of its dependencies is not always appropriate or even possible. The reason are three folds. First of all, some software dependencies such as web services and databases are very slow when they are invoked. Thus involving such dependencies in testing will slow down the whole testing process, which may be fine for system testing, but not acceptable in unit testing and regression testing which are typically performed whenever a change is committed. Second, in unit testing, the testers should focus on the unit (e.g., functions, classes) to be tested. Involving some unstable dependencies may result in test interference due to bugs in the dependencies, and make it more difficult to identify and fix bugs inside the unit. Third, in large software projects, it is common that software developers develop different modules simultaneously and integrate the modules later. In such cases, it is probable that testing a certain module requires another module that has not been developed yet. Also, when developing frameworks that support call-backs, it is common that the dependencies invoked by the call-backs are not available at the testing phase (the call-backs are supposed to be defined by users of a framework).

In the practice of testing object-oriented software development process, to resolve the issues brought by software dependencies, mock objects are a typical solution. Primitive mock objects are

also known as server stubs, introduced in Binder's book [19] as a code pattern for testing object-oriented systems. Mock objects can be used to replace real software dependencies by simulating the relevant features of software dependencies. Typically, methods of mock objects are designed to return some desired values (either a constant value for all inputs, or different values according to different inputs) when specific input values are given. Furthermore, to guarantee the correct interactions between the software component under test and dependencies, mock objects may check their interactions with the software component under test. For example, a mock email sender object of an email client may check whether the email-sending method is ever invoked by the email client and whether the arguments passed into the mock object are in correct format.

Recently, to facilitate developers in developing mock objects, a number of mocking frameworks (e.g., Mockito, EasyMock) have been developed. These mocking frameworks provide APIs for creating new mock objects, setting return values of methods in the mock objects, and checking interactions between the component under test and the mock objects. Along with the usage of mock objects and mocking frameworks, there are also many debates among software developers and testers on their usages. For example, some developers argue that mock objects should be used whenever a dependency is invoked because involving real dependencies may result in bug interference, but other developers do not agree because involving mock objects will also bring in a lot of overhead in test execution and maintenance. Also, developers debate on whether mocking frameworks should be used for all mock objects, and what features of mocking frameworks are most useful.

Despite the development of various mocking frameworks, and debates about usage of mock objects and mocking frameworks, there are very few academic empirical studies to observe and understand how mocking frameworks are used in software testing practice. In this paper, we present an empirical study to fill this blank area. In our study, we analyze the usage of mocking frameworks in 5,000 open source software projects from GitHub[1]. Specifically, we studied the popularity of mocking frameworks in the testing of open source software projects and how devel-

[1]GitHub: http://github.com

opers of open-source software projects are using the frameworks. Our study finds a substantial usage of mocking frameworks and mock objects among open source Java software projects, but software testers usually mock only a small number of software dependencies. We also find that the unique features of certain mocking framework are frequently used, and there are more mocking of source code classes than mocking of library classes.

The main contributions of this paper are:

- We carried out a large scale empirical study on more than 5,000 open source software projects from GitHub.
- We discover a number of interesting facts about the usage status of mocking frameworks, including their popularity, most frequently used features, and the characteristics of mocked objects.

In Section 3.2 and Section 3.3, we present the design details, and the study results of our empirical study, respectively. Before we conclude in Section 3.5, we introduce some related research efforts in Section 3.4.

3.2 Design

In this section, we introduce the design of our empirical study on the usage of mocking frameworks among developers of open source software projects.

3.2.1 Research Questions

In this study, we design an experiment to answer the following research questions. By answering these questions, we try to understand whether and how software developers and testers use mocking frameworks to handle dependencies in the testing of open source software projects.

- **RQ1:** How popular are mocking frameworks that are used in the testing of open source Java software projects? Are developers trying to mock most or all of dependencies?

Table 3.1: Basic Information of Subjects Used in Study

Info Type	Avg.	Med.	Max.	Min
# Java Files	153	18	10,141	1
LOC	25,305	1,833	3,094,265	0
# Test Files	16.4	0	3,335	0
# Developers	3.0	1	39	1
# Versions	11.4	1	1694	1

- **RQ2:** What features of mocking frameworks are most frequently used in the testing of open source software projects?

- **RQ3:** What types of dependencies developers tend to mock?

3.2.2 Subjects

To perform our empirical study, we randomly downloaded 5,000 Java software projects from Github. We choose to focus on Java software projects because Java is one of the most widely used object-oriented programming language, and due to the popularity of Java in the open source community, a large number of open source Java software projects may serve as subjects. When selecting subject projects, we first search Java projects in Github with 26 keywords from 'a' to 'z', and then, we randomly sample 5000 projects from the merged search results. For each project, we consider only the most recent version.

The basic information of the 5,000 Java software projects are presented in Table 3.1.

In Table 3.1, we present five types of basic information of the collected software projects, which are in the Lines 2-6 of the table. Specifically, Line 2 presents the number of Java Files in the project. It should be noted that some of the software projects may be developed in multiple programming languages and may involve other types of source code files. In our study, we focus only on the Java part of such software projects. Line 3 presents the total lines of source code in the Java source files. Line 4 presents the number of test files in the software projects. Specifically, we consider a Java source file as a test file if it uses any library classes from JUnit or TestNG, which are the two major test frameworks for Java and the only test frameworks we notice the collected software projects are using. It should be noted that, in some software projects, a test class C may

extend a library class in test frameworks, and other test files use only C but not any library classes in test frameworks. So in our study, we also consider the test files that indirectly use library classes in test frameworks by using classes like C. Line 5 and Line 6 present the number of developers and the number of versions (i.e. code commits) of the software projects.

Due to the large number of subjects, we are not able to present the basic information of individual software projects. Instead, we present the statistical information of the software projects. Column 2-5 present the average, medium, maximum and minimum of all types of information, respectively.

From Table 3.1, we can see that the chosen subjects vary largely in their sizes, testing efforts, number of developers, and history. The average size of the subjects are 25KLOC, and the largest subject has more than 3 million lines of code. Furthermore, more than half of the subjects have no test code (the medium of number of test files is 0), and the average number of test classes is 16.4.

3.2.3 Process

To answer the research questions listed in Section 3.2.1, for each of the collected subject, we first identify all the Java source files and test files. Then, we parse these files to extract the code relevant to mocking frameworks.

In particular, we consider the four most popular Java mocking frameworks in our study: Mockito, EasyMock, JMock, and JMockit. PowerMock is another popular mocking framework, but it is actually an extension of Mockito and EasyMock frameworks, and typically software projects using PowerMock will use at least one of Mockito or EasyMock frameworks, so we do not consider PowerMock in our study.

The four investigated frameworks have some specific characteristics. EasyMock requires developers to specify the type of the mock object at the creation point (i.e., whether the mock object is strict or not, and whether to raise exceptions when there are unexpected methods invoked). Typically, EasyMock is more strict to raise more run time errors, and the interface is not very easy to learn (comments from StackOverflow). Mockito actually origins from EasyMock, and it provides a very simple API interface to create mocking objects (not strict by default) and support annotations

to specify the classes to be mocked. Mockito also provides very good spy features that verifies the dependency-related interaction of the class under testing on the real dependency classes. JMock is similar to EasyMock in most aspects. JMockit uses class-map as its underlying technique for mocking and thus can mock final classes.

From the relevant code, we extract the information that can help answer our research questions, Specifically, we extract a data set of API signatures of each mocking framework from their API documents, and developed a AST-tree-based code analyzer analyze the source code of subject projects extract all relevant API invocations. Based on these API invocations, we are able to identify the API method in mocking frameworks used, the dependency classes being mocked, etc.

3.3 Empirical Study Results

In this section, we present the details and results of our empirical study. Specifically, we present the study to answer the research questions 1 to 3 in Section 3.3.1, Section 3.3.2, and Section 3.3.3, respectively. We summarize our findings in Section 3.3.4, and present the threats to validity in Section 3.3.5.

3.3.1 Popularity of Mocking Frameworks

To answer the first research question, we study the number of software projects using mocking frameworks among the 5,000 downloaded software projects. Specifically, among the 5,000 software projects, 2,046 has at least one test class (a class importing and use any class from JUnit or TestNg). Among the 2,046 projects, 459 projects (23%) uses at least one mocking framework. We guess that developers of large software projects tend to use mocking frameworks, so we compare the size distribution of software projects using mocking frameworks with the size distribution of all software projects having test code, and present the results in Figure 3.1. The figure shows that the size of software projects using mocking frameworks tends to be larger than other software projects with test code. Specifically, the median size of software projects using mocking frameworks is 16KLOC, compared to 8.4KLOC of all software projects with test code (software projects in the

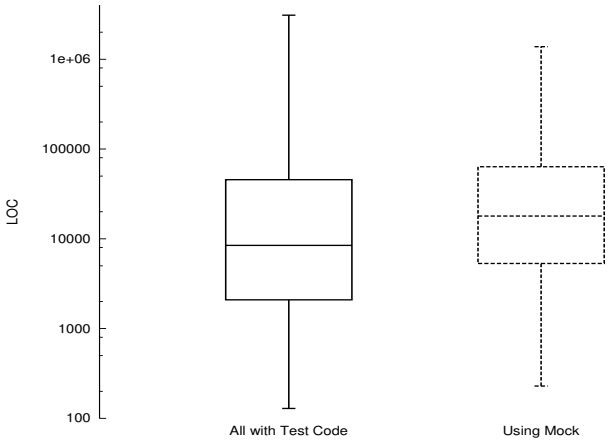

Figure 3.1: Size comparison of software projects using mocking frameworks and all software projects with test code

latter set include software projects in the former set). At the same time, we also found that there are many large software projects do not use any of the investigated mocking frameworks. Furthermore, among the software projects (with test code) that are larger than 10KLOC, and 100KLOC, the proportion of software projects using mocking frameworks goes up to 30% and 34%, respectively.

On top of the popularity of mocking frameworks in open source software projects, we are also interested in the market share of various popular mocking frameworks. Therefore, we studied the usage of different mocking frameworks and presented the results in Table 3.2. From the table, we can see that, among the four major mocking frameworks, Mockito is the most widely used and is used in more than 70% of software projects that use mocking frameworks. EasyMock and JMock rank second and third, and are used in about 20% and 10% of software projects using frameworks, respectively. JMockit is the least widely used among the four. It should be noted that this observation may not be generalized because it is from 5,000 randomly selected Java projects from GitHub.

Furthermore, we found that 45 software projects use more than one mocking frameworks. We further studied the projects and found that the main reason for software developers to use multiple mocking frameworks is that, different software testers are more familiar with different mocking frameworks so that they use different mocking frameworks in the testing. Another reason is that

Table 3.2: Popularity of Mocking Frameworks

Mocking Framework	# Projects
Using Mockito	340
Using EasyMock	106
Using JMock	45
Using JMockit	7
Total	459
Multiple Mocking Frameworks	45

Table 3.3: Usage of Mocking Frameworks in Test Classes

	Avg.	Med.	Max.	Min
# Test Files	16.8	5	1037	1
Proportion of Test files	29.8%	20%	100%	0.3%

the software project is moving from one framework to another.

We observed that mocking frameworks are used widely in software projects. However, the data above just reveals whether a software project involves mocking frameworks in any test class. It is still not known whether mocking frameworks are used in many test classes or only a small number of test classes. To answer this question, we further studied the number and proportion of test classes that use mocking frameworks and present the results in Table 3.3

From the table, we can see that, in software projects that use mocking frameworks, on average, 16.8 test classes are using mocking frameworks, accounting for about 29.8% of all test classes. This observation shows that software testers are not using mock objects for all dependencies. Also, it is not the case that mock objects are used very rare and in only some special cases.

Moreover, we studied the number of mock objects in one test class and present the results in Table 3.4. From the table, we can see that most test classes (77.2%) have created 3 or less mock objects. Second, we try to understand, for test class that uses mocking frameworks, how many dependency classes are mocked and checked. Specifically, we deem a test class as dependency class if it is imported by the test class. The results are presented in Table 3.5. The table shows that on average about 17% of dependency classes are mocked by the software testers. Thus, software testers seem to mock only a small portion of all dependency classes of a test class.

Table 3.4: Number of Mock Objects in Test Classes

# Mock Objects	# Test Files	Proportion
1	1270	45.3%
2	587	21.0%
3	333	11.9%
4	210	7.5%
5	135	4.8%
6	73	2.6%
7	49	1.7%
8	40	1.4%
9	22	0.8%
10+	82	2.9%

Table 3.5: Proportion of Mocked Classes

	Avg.	Med.	Max	Min
# Mocked Classes	2.7	2	40	1
# Dependency Classes	17.0	14	215	1
Proportion of Mocked Clases	17.8%	14.3%	100%	0.9%

3.3.2 Usage of Mocking Frameworks

To answer the second research question, we try to study what APIs in mocking frameworks are most widely used. Since different mocking frameworks have different API sets, in this phase, we study only Mockito and EasyMock, which are the most and second most widely used mocking frameworks. The results are presented in Table 3.6 and Table 3.7

From Table 3.6 and Table 3.7, we have the following two main observations. First of all, verification of mock objects are extensively used. This fact shows that software testers do not use mocking frameworks simply as a convenient tool for generating normal test stubs or fake objects, but are using the core features of mock objects (i.e., specifying and verifying interactions between the component under test and the software dependency). Second, several advanced APIs in Mockito are widely used. Such APIs include “spy” which partially mock an existing class (i.e., use the mock object for specified method invocations, but the real dependency for other method invocations), “times” that specify how many times a certain method is invoked, etc.

Table 3.6: Most popular ten APIs in Mockito

API	Frequency	Description
Mockito.verify	2249	verify tester's specification of the mock object
Mockito.mock	1678	create an empty mock object
Mockito.when	1471	specify a method in the mock object to be invoked
Mockito.spy	403	partially mock an existing class
Mockito.times	232	specify how many times a method is invoked
Mockito.given	161	Alias of "when" for behavior driven development style
Mockito.never	152	specify that a method is never invoked
Mockito.verifyNoMoreInteractions	98	specify that there are no more interactions between the component under testing and the mock object
Mockito.doReturn	85	specify the return value of a method invocation
Mockito.verifyZeroInteractions	74	specify that there are no interactions between the component under testing and the mock object

Table 3.7: Most popular ten APIs in EasyMock

API	frequency	description
EasyMock.expect	671	specify a method in the mock object to be invoked
EasyMock.createmock	647	create an empty mock object
EasyMock.replay	638	notice the mock object that the testing of component under testing starts
EasyMock.verify	510	verify tester's specification of the mock object
EasyMock.expectLastCall	144	alias for "expect" for void methods
EasyMock.eq	90	expect an value that is equals with a given value
EasyMock.createNiceMock	67	create a mock object that check the order of method invocations
EasyMock.anyObject	64	match with any type of arguments when specifying the invocation of a method
EasyMock.reportMatcher	58	specify a matcher of arguments for specification of method invocations
EasyMock.capture	53	capture a matched value for later access

Table 3.8: Mocking of Library Classes

	Avg.	Med.	Max	Min
# Mocked Library Classes	0.85	0.0	18	0
Proportion of Library Classes	39.4%	0%	100%	0%

Table 3.9: Mostly Mocked Library Classes

Mocked Library Class	Frequency
javax.servlet.http.HttpServletRequest	44
org.fest.assertions.description.Description	41
javax.servlet.ServletContext	33
javax.jcr.Node	32
rx.Observer	24
org.openqa.selenium.WebDriver	23
javax.servlet.http.HttpServletResponse	22
javax.jcr.Property	20
javax.jcr.Session	20
java.io.InputStream	17

3.3.3 Mocked Dependencies

To answer the third research question, we studied the classes that are mocked by software testers using mocking frameworks. Specifically, we try to find out whether software testers tend to mock library classes and what are the mostly mocked library classes. To find the answer, we check whether the mocked classes are library classes (i.e., classes that are not in the source code base of the software project), and present the results in Table 3.8.The results show that about 39% of all mocked classes are library classes, so software testers actually tend to mock more classes in the source code, compared with library classes. This shows that the major reason for software testers to perform mocking may be parallel development, instead of accelerating testing process and verifying interactions of classes with dependencies.

In Table 3.9, we present the top 10 mostly mocked library classes. Specifically, the frequencies are the times that the specific class is mocked in all the studied open source software projects. From the table, we can observe that the two major source of mostly mocked classes are the package “javax.servlet”, and the package “javax.jcr”. Specifically, the former package is about HTTP request and responses, and the latter package is about the operation of Java Content Repositories.

3.3.4 Summary

To sum up, the major findings of our empirical study are as below.

- Mocking frameworks and mock objects are used widely among open source Java software projects. However, software testers usually mock only a small number and portion of software dependencies.

- Verification is widely used to check the specified interaction between the component under testing and the mocked object. Special APIs in both Mockito and EasyMock are widely used, which implies an incompleteness of features for a single mocking framework.

- Software testers tend to mock source code classes than libraries, while library classes also take a substantial proportion (40%) in all mocked classes. The mostly mocked library classes are classes from packages handling HTTP requests / responses, and content repositories.

3.3.5 Threats to Validity

The major threat to the internal validity of our study is the potential errors in the programs analyzing the collected software projects and performing statistics. To reduce the threat, we tried our best to write the code carefully to avoid any bugs in the programs. The major threat to the external validity of our study is that our observations may be specific to the subject software projects, and cannot be generalized other software projects. To reduce this threat, we use a large number of subject software projects, and choose the subjects randomly. Also, it is possible that our findings are specific to Java software projects. To further reduce this threat, we plan to carry out similar empirical studies on subjects written with other major programming languages in the future.

3.4 Related

As far as we know, this paper is the first research effort to study the usage status of mocking frameworks and mock objects in software practice. There have been a number of existing research

efforts on enhancing mocking frameworks or leveraging mock objects to improve automatic software testing. Freeman et al. [45] [44] presented the basic process and concepts of using mocks objects in unit testing, as well as a mocking framework jMock. Galler et al. [46] proposed an approach to automatically generation of mocking objects satisfying certain preconditions to serve as test inputs in automatic test-case generation for Java unit testing. Taneja et al. [117] proposed to automatically generate mock objects to simulate the behavior of database systems in the automatic test-case generation of database systems. Coelho et al. [29] proposed an approach to generate mock agents in the unit testing of multi-agent software systems. Due to the necessity of mock objects in unit testing, researchers also proposed approaches to automatically generating mock objects along with unit test cases [57] [99]. For the existing test cases which are not using mock objects, Saff and Ernst [112] proposed an approach to automatically refactor such test cases by adding mock objects. Marri et al. [81] carried out an experiment to study the benefit of using mock objects to simulate file systems. All these research efforts are about automatic generation of mock objects and how to leverage mock objects in specific testing problems, while our work focuses on the current usage status of mocking frameworks and mock objects in real world software projects.

There are also some existing studies investigating the status of testing practice, or the effectiveness of testing techniques on real-world open source projects. Singh et al. [100] empirically explored 20,000 open-source projects, and found that bigger projects have a higher probability to contain test cases and projects with more developers may have higher number of test cases. In addition, they also found that the number of test cases has a weak correlation with the number of bugs. Greiler et al. [50] explored the current testing practices currently used for the specific plug-in systems. Pham et al. [103] investigated how social coding sites influence testing behavior. They found several strategies that software developers and managers can use to positively influence the testing behavior. Fraser et al. [43] empirically evaluated automated test generation on 100 real-world Java projects. They found that high coverage is achievable on commonly used types of classes, and also identified future directions for further improve code coverage. In this paper, we aims to investigate the differences between manual tests and DSE-based tests for real-world

systems.

3.5 Conclusion

In this paper, we direct a large scale empirical study on the usage of mocking frameworks in software testing. To perform the study, we collected 6,000 Java software projects from Github, and analyze the source code of the projects to answer a number of questions about the popularity of mocking frameworks, the usage of mocking frameworks, and the mocked classes. Our major findings include that mocking frameworks are widely used in practice, and a small portion of dependencies are mocked. This finding shows the requirement of more research on mocking frameworks, as well as on the reasons why developers choose to mock a class while not to mock the other. Furthermore, we find that a number of unique features in Mockito and EasyMock are widely used. This implies that it is possible to build better mocking frameworks by incorporating the most popular features of existing mocking frameworks. We also find that software developers tend to do more mocks on source code classes than library classes, which is not as we expected.

In the future, we plan to extend our work in the following directions. First of all, we plan to direct similar studies on software projects written in programming languages other than Java to check whether our findings can be generalized. Second, we plan to develop techniques to help software developers and testers make decisions on whether or not a class should be mocked. Third, we plan to develop techniques to generate mock objects automatically with mocking frameworks in automatic unit testing.

Chapter 4: PERFRANKER: PRIORITIZATION OF PERFORMANCE REGRESSION TESTS FOR COLLECTION-INTENSIVE SOFTWARE

4.1 Introduction

During software evolution, frequent code changes, often including problematic changes, may degrade software performance. For example, a study [56] found that upgrading from MySQL 4.1 to 5.0 caused the loading time of the same web page to increase from 1 second to 20 seconds in a production e-commerce website. Even small performance degradation may result in severe consequence. For example, Google could lose 20% traffic due to an increase of 500ms latency [82]. Amazon could have 1% decrease in sales due to a 100ms delay in page rendering [116].

Developers can apply systematic, continuous performance regression testing to reveal such performance regressions in early stages [31,42,61,88,124]. But due to its high overhead, performance regression testing is expensive to conduct frequently. Actually, the typical execution cost of popular performance benchmarks varies from tens of minutes to tens of hours [56], so it is impractical to run all performance test cases for each code commit. Recently, PerfScope [56] was proposed to predict whether a code commit may significantly affect software performance and thus require performance testing. Specifically, PerfScope extracts various features from the original version and the code commit, and trains a classification model for prediction. Although PerfScope helps reduce code commits for performance regression testing, its empirical evaluation shows that a non-trivial proportion of code commits still require performance testing; thus, there is still a strong need of reducing the cost of conducting performance regression testing on a code commit, even after applying PerfScope in practice.

To address such strong need, developers shall prioritize performance test cases on a code commit for three main reasons. First, there can be high cost to execute all performance test cases on a

code commit for large systems in practice. Second, as reported in a previous industrial study [124] and our study in Section 4.2, various random factors may affect the observed execution time, so it typically requires a large number of repetitive executions to confirm a performance regression. Therefore, with prioritized test cases, developers can better distribute testing resources (i.e., do more executions on test cases likely to trigger performance regressions). Third, a code commit may accelerate some test cases while slowing down others. It is often important for the developers to understand the performance of their software under different scenarios, while a coarse-grained commit-level technique is not helpful on this requirement.

To develop an effective test-prioritization solution for performance regression testing, we focus on *collection-intensive software*, an important type of modern software whose execution time is heavily spent on loading, manipulating, and writing collections of data. Collections are widely used in software for scalable data storage and processing, and thus collection-intensive software is very common. Examples include libraries for data structures, text formating and parsing, mathematics, image processing, etc. Also, collection-intensive software is often used as components in complex systems. Moreover, a recent study [59] shows that a large portion of performance bugs are related to loops, which are often used to iterate through collections. Our statistics show that 89% and 77% of loops iterate through collections for our two subjects Xalan and Apache Commons Math, respectively.

For collection-intensive software, a straightforward approach to prioritizing performance test cases on a code commit would be measuring collection iterations (e.g., loops) impacted by the code commit and executed by each test case. However, such an approach may not be precise enough to differentiate test cases in the presence of newly added iterations, manipulations, and processing of collections, as well as their effect on existing collection iterations. Consider the simplified code example from Xalan in Listing 4.1. The code commit involves a new loop, and its location may or may not be at the hot spot for all or most test cases. Therefore, its impact on different test cases

may largely depend on the different iteration counts of the added loop, the side effect of changing variable `list`, and the operations in the loop. Since Loop_B depends on a collection variable `limits`, which further depends on Loop_A, we can infer the test-case-specific iteration count of Loop_B from that of Loop_A; such iteration count can be acquired by profiling the base version for all test cases. Furthermore, we can infer the effect of adding `list.add(...)` on `list` with the iteration count of Loop_B, and update the iteration count of loops dependent on `list`. Moreover, we can enhance the estimation precision by using test-case-specific execution time of operations (e.g., `new Arc(...)`).

```
1     while(i <= m\_size){ //Loop A
2       limits.add(new Limit(...))
3       ...
4     }
5     ...
6   + Collections.sort(limits);
7   + for (int i = 0; i < limits.size()-1; i++) { //Loop B
8   +   list.add(new Arc(limits.get(i),  ...));
9   + }
```

Listing 4.1: Collection Loop Correlation

These observations inspire us for three main insights to effectively model a code commit and its effect on existing collections and their iterations. First, collection sizes (e.g., `limits.size()`) and loop-iteration counts (e.g., Loop_B) can often be correlated, so collection sizes can be inferred from loop-iteration numbers and vice versa. Also, collection variables (e.g., `limits`) can be used as bridges to infer iteration counts of new loops (e.g., Loop_B) from existing loops (e.g., Loop_A). Second, collection manipulations (e.g., `list.add(...)`) are often inside loops, so the size of collections referred by collection variables (e.g. `list`) can be estimated from loop-iteration counts (e.g., Loop_B). Third, due to the large number of elements in collections, the average processing time of elements (e.g., `new Arc(...)`) is relatively stable, so a method's average execution time in the new version may be estimated from that in the base version.

Based on the three insights, we propose PerfRanker, which consists of four automatic steps. First, on a base version, we execute each test case in a profiling mode to collect information about the test execution, including the runtime call graph and the iteration counts of all executed loops.

We also perform static analysis to capture the dependency among collection objects and loops. Second, based on the profiling information, we construct a performance model for each test case. Third, given a code commit, we estimate the execution time of each test case on the new version (formed by the code commit) by extending and revising its old performance model. We use profiling information and loop-collection correlations to infer parameters of the new performance model, and refer to this step as *Performance Impact Analysis*. Fourth, we rank all the test cases based on the performance impact on them.

We implement our approach and apply it on two sets of code commits collected from popular open source collection-intensive projects: Apache Commons Math and Xalan. To measure the effectiveness of test case prioritization for a code commit in performance regression testing, we use three metrics: (1) *APFD-P* (Average Percentage Fault Detected for Performance), an adapted version of the APFD metric [80] for performance testing, (2) *DCG* [11], a general metric for comparing the similarity of two sequences, and (3) *Top-N Percentile*, which calculates the percentage of test cases needed to be executed to cover the top N test cases whose execution time is most affected by the code commit. Our evaluation results show that, compared with the best of the three other baseline approaches, our approach achieves an average improvement of 17.6 percentage points on *APFD-P* and 27.4 percentage points on DCG. Furthermore, for Apache Commons Math and Xalan, our approach is able to rank top 1 affected test case within top 8% and top 16% test cases, and top 3 affected test cases within top 37% and 30% test cases, respectively.

This paper makes the following major contributions:

- A novel approach to prioritizing test cases in performance regression testing of collection-intensive software.
- Adaptation of the APFD metric to measure the result of test prioritization for performance regression testing.
- An evaluation of our approach on real-world code commits from two popular open source collection-intensive projects.

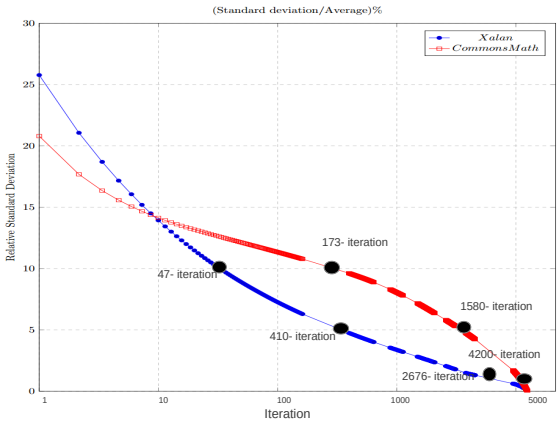

Figure 4.1: Relative Standard Deviation vs. Sample Size

4.2 Motivation

In this section, we provide preliminary study results to motivate prioritization of performance regression tests, due to high cost of executing the same test case for many times for performance regression testing. In particular, modern mechanisms in hardware and software often bring in random factors impacting performance. Some well-known examples are the randomness in scheduling cores and buses in multi-core systems [98], in caching policies [105], and in the garbage-collection process [125], etc. These factors interact with each other and amplify their effect so that the execution time of a test case may vary substantially from time to time.

To neutralize such randomness, researchers or developers execute a test case multiple times and calculate the average performance [77]. To better understand this requirement, we perform a motivating study (with more details on the project website [8]) on the two open source projects used in evaluating our approach. In particular, we execute the test cases for 5,000 times, and randomly select samples with different sizes to calculate their standard deviation. In Figure 4.1, we show how the execution time's relative standard deviation (y axis) changes as the execution times (x axis) increase from 1 to 5,000. The figure shows that the average relative standard deviation with sample size 1 is over 20% in both projects. In other words, if only one execution is used, the recorded execution time is expected to have more than 20% difference from the average execution time in the 5,000 executions. Note that 20% is a very large variance because 10% performance

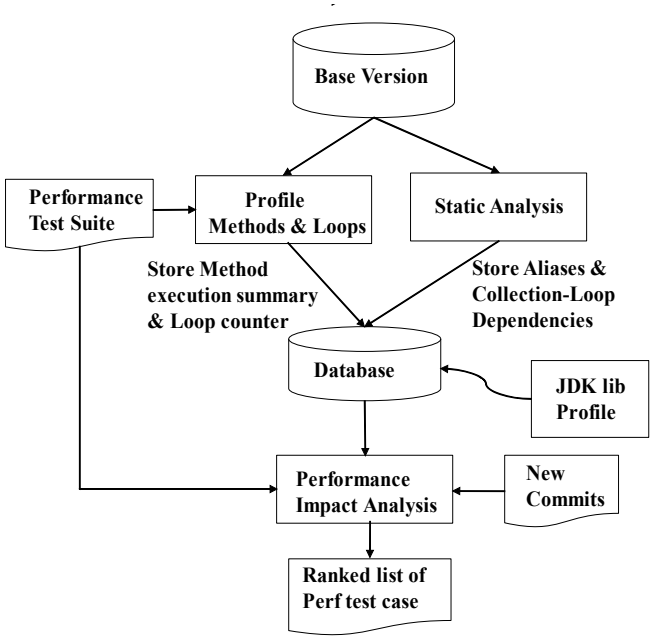

Figure 4.2: Workflow of Our Approach

enhancement is typically considered significant, and techniques incurring over 10% overhead are often considered too slow for deployment purposes [76]. The figure also shows that 173 and 47 executions are required to achieve relative standard deviation less than 10% for Apache Commons Math and Xalan, respectively. Executing the whole test suite for many times can be prohibitively expensive, calling for prioritization of performance regression tests, as targeted by our approach.

4.3 Approach

In this section, we introduce our test prioritization approach in detail. In particular, we first present the overview of our approach, major technical challenges, and our performance model. After that, we introduce performance-impact estimation of a code commit based on our performance model, including method-execution-time estimation, and method-invocation-frequency estimation based on collection-loop correlation and iteration-count inference.

4.3.1 Overview

Figure 4.2 shows the workflow of our approach. The input to our approach includes the project code base, its code commits, and performance test cases. The output of our approach is an ordered list of performance test cases. The first step of our approach is to profile the base version of the project under test. During the profiling, for each test case, we record the dynamic call graph as its original performance model. We also record average execution time and frequency of methods, and iteration counts of all loops. At the same time, we statically analyze the base version to gather dependencies between loops and collection variables, as well as aliases among collection variables. When a new code commit comes, we conduct performance impact analysis to estimate its performance impact on all test cases, and prioritize test cases accordingly.

Technical Challenges. Performance impact analysis is the core of our approach. Although we focus on collection-intensive software, it is still challenging to conduct performance impact analysis, facing three major technical challenges:

- Challenge 1. A code commit may include any type and scope of code changes, from one-line revision, to feature addition and interface revision. Therefore, there is a strong need of a unified and formal presentation for code commits.

- Challenge 2. A code commit may contain newly added code, especially new loops. No execution information of such code is available, but given that loops can have high impact on performance, there is a strong need of estimating the code commit's execution time and frequency.

- Challenge 3. Even if the execution time of changed code in a code commit has little impact on performance, the code commit may include changes on collection variables, eventually affecting the performance of unchanged code.

To address Challenge 1, we present a code commit as three sets of methods: added methods, revised methods, and removed methods. As our performance model is based on the dynamic call

graph, any code commit can be mapped to a series of operations for method addition, removal, and replacement in the performance model. To address Challenge 2, we leverage the recorded profiling information of the base version as much as possible. Specifically, if an existing method is invoked in the newly added code, we can use the recorded execution time of the existing method as its execution-time estimation for this new invocation. Furthermore, as discussed in Section 5.1, we use collection variables as bridges to estimate iteration counts of new loops from those of existing loops. To address Challenge 3, we track all the element-addition and element-removal operations of collection variables in the newly added code, and estimate the size change of collection variables from the iteration count of their enclosing loops. This new size is used to update the iteration counts of loops depending on the changed collection variables.

Since we focus on collection-intensive software, we consider only loops whose iteration number depends on collection variables, e.g., variables of array type, and other collection types defined in Java Utility Collections. Note that there are also some loops whose iteration number depends on simple integers, such as a loop to sum up a numbers from i to j, but such loops are not common in collection-intensive software, and our evaluation results show that our approach is effective on both data-processing software (Xalan) an mathematics software (Apache Commons Maths).

4.3.2 Performance Model

In this subsection, we introduce our performance model to break down execution time of a test case to all the methods invoked by the test case. The basic intuition behind our model is that the execution time of a method invocation M is the execution-time sum of all method invocations directly invoked in M, together with the execution time of instructions in M. Since most basic operations in Java programs are performed by JDK library methods (e.g., a string concatenation), the latter part is typically trivial compared with the former part, so our performance model ignores instructions in M itself, but focuses only on methods that M invokes.

We illustrate our model in Figure 4.3, where each node represents a method and each directed edge represents an invocation relation. Here each node annotated with label t_{avg}, which represents

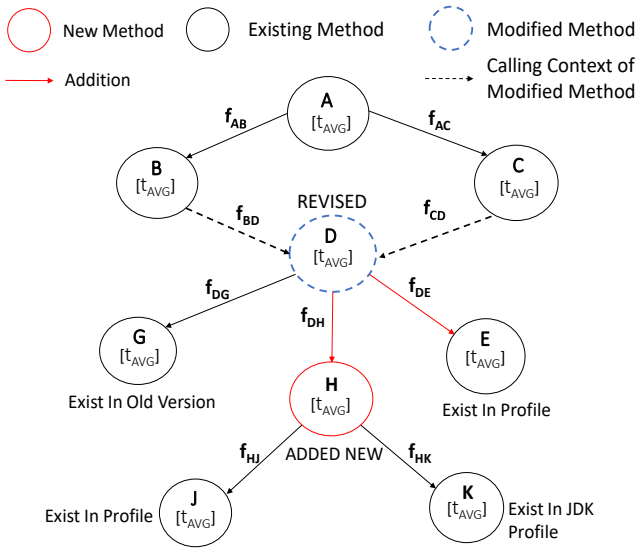

Figure 4.3: An Example Performance Model

the average execution time of a method. Each edge in the graph is labeled with f_{AB}, which represents the invocation frequency of method B from method A. Given a code commit, the performance model of the post-commit version can be acquired by adding and removing nodes and edges to the original performance model. For example, in Figure 4.3, method D is a revised method and it now calls (1) method G, which it originally calls, (2) E, which is an existing method in the base version, and (3) H, which is a newly added method. With the average execution time and invocation frequency of all invoked methods in D, we are able to calculate an execution-time estimation for revised method D. The new execution time at D can be propagated upward to its ancestors, until the main method is reached and a new estimation of the whole program's execution time can be made.

4.3.3 Performance Impact Analysis

The basic idea of performance impact analysis is to calculate the execution-time change of each revised method M in the code commit. Then, through propagating the execution-time change to M's predecessors in the performance model, we can calculate the execution-time change of the whole test case at the root node.

We realize this idea in three steps. First, for each revised method, we extend the performance model to either add it and/or some of its callees (and transitive callees). Second, we estimate the execution-time change of each method in the new performance model. Third, we estimate the invocation frequency on the edges of the new performance model. We next introduce the three steps in details.

Model Extension

For each revised method, we add its direct and transitive callees (e.g., methods E through K in Figure 4.3 for the revised method m_D) into the performance model, if they do not already exist in the model. In this recursive process, we terminate the extension of a method node if it is an unrevised method existing in the base version, a JDK library method, or a method whose source code is not available. Since we use one base version for a series of code commits, a revised method (and even some of its predecessors) may not exist in the base version because they are added after the base version. In such a case, we transitively determine M's predecessors (callers) until we reach methods in the base version. For example, if C and D in Figure 4.3 are added between the base version and the code commit under analysis, we determine that D is invoked by B and C, and C is invoked by A, so that we add C and D to the performance model.

When the new code version of the revised methods is available, we statically determine the direct and transitive callers and callees for the revised method, and one remaining challenge is to resolve polymorphism, where one method invocation may have multiple targeted method bodies. Although it is straightforward to apply off-the-shelf points-to analysis, since we have the profiling information of the base version, we make use of the information to acquire a more precise call graph. Specifically, if a method invocation is not involved in the method diff (i.e., the method invocation can be mapped to the same method invocation in the base version, such as G in Figure 4.3), we assume that its target is not changed and we use the same targeted method body as recorded for the base version. Otherwise, we apply points-to analysis [73] in Soot [120] to find the possible targeted method bodies for the method invocation. When a method invocation is mapped to

multiple method bodies, we add all bodies to the new performance model, and we divide the estimated frequency of the method invocation by the number of possible targets to attain the invocation frequency of each target.

Execution-Time Change of Method Bodies

The method bodies invoked from a revised method fall into three categories. The first category is removed method bodies. Their execution-time data are recorded in the performance model of the base version, and their new execution time is estimated as 0.

The second category includes method bodies already existing in the performance model of the base version. Such method bodies include both those defined in the source code and those defined in the JDK library, or those without source code. For existing method bodies, we simply use the recorded average execution time in the base version as their estimated average execution time. For bodies of JDK library methods, we profile Dacapo [21] to acquire the average execution time of those common JDK library methods. For methods without source code or those not invoked by Dacapo, we use the average execution time of all method bodies in the profile as their estimated execution time, as we have no further information. Note that when a method from the second category is added to the performance model, its original execution time is set as 0.

The third category includes newly added method bodies in the source code. *Note that such method bodies include both those added to the source code in the code commit and those added in any other code commits between the base version and the code commit under analysis.* They also include method bodies defined in libraries but are newly reached due to code revisions from the base version to the new version. For a newly added method body (such as H in Figure 4.3), as discussed in Section 4.3.3, we extract all its callee method bodies, and add them to the performance model (such as J and K in Figure 4.3), and then we calculate its execution-time change using our performance model.

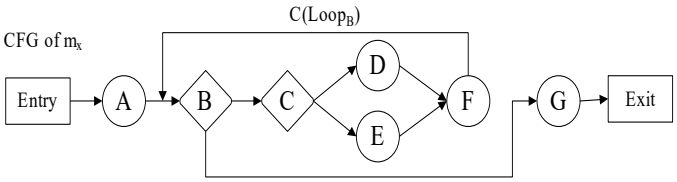

Figure 4.4: An Example Control Flow Graph

Invocation Frequencies of Method Bodies

Given a newly added or revised method m_x, for each method m_y that is directly invoked in m_x, we estimate the invocation frequency of m_y in m_x (denoted as $fq(m_x, m_y)$) to apply our performance model on the new version. Our estimation technique is based on the control flow graph of m_x and the average iteration counts of loops in m_x. Specifically, for any code block b in m_x, we use $fq(b, m_y)$ to denote the invocation frequency of m_y from b for each execution of b. If b is a basic block without branches and loops, $fq(b, m_y)$ is exactly the number of invocation statements to m_y in b; such number can be easily counted statically. Then we calculate $fq(m_x, m_y)$ by applying the inference rules for sequential, branch, and loop structures in Formulas 4.1-4.3 below recursively on the code blocks of m_x:

$$fq([b_1; b_2], m_y) = fq(b_1, m_y) + fq(b_2, m_y) \tag{4.1}$$

$$fq([\text{if() } b_1 \text{ else } b_2], m_y) = Max(fq(b_1, m_y), fq(b_2, m_y)) \tag{4.2}$$

$$fq([\text{while}_i \text{ () } b]) = fq(b, m_y) \times C(loop_i) \tag{4.3}$$

In the inference rules, the only unknown parameter is $C(loop_i)$, which denotes the average iteration count of the ith loop in A. As an example, given the control flow graph in Figure 4.4 of m$_x$, for any m_y that m_x invokes, $fq(m_x, m_y)$ can be estimated as in Formula 4.4.

$$fq(m_x), m_y) = fq(A, m_y) + (Max(fq(D, m_y), fq(E, m_y)) + fq(F, m_y)) \times C(Loop_B) + fq(G, m_y) \tag{4.4}$$

4.3.4 Loop-Count Estimation with Collection-Loop Correlation

With the estimation of invocation frequency, the only remaining unknown parameter in the performance model of the new version is the loop count of all loops. If a loop exists in the base version and is not affected by the code commit, we directly use the recorded profile from the base version to acquire the iteration count. Two more complicated cases are (1) when a new loop is added, and (2) when the code commit affects the iteration count of an existing loop. Here is our insight: for collection-intensive software, we can construct the correlation between collection sizes and loop counts, and use iteration counts of known loops to infer that of unknown loops, as well as a code commit's impact on iteration counts of known loops.

Correlating Loops and Collections

In particular, we consider the following two types of dependencies between loops and collection variables:

- Iteration Dependency. A loop L is iteration-dependent on a collection variable v if L's loop condition depends on the size attribute of v.
- Operation Dependency. A collection variable v is operation-dependent on a loop L if there exists an element addition or removal on v in L.

To identify iteration dependencies, for a For-Each loop (e.g, `for(A a : ListOfA)`), we simply consider that the loop is iteration-dependent on the collection variable being iterated via the loop. For other loops, we use standard inter-procedural data flow analysis [22] [108] to track data dependency backward from the loop condition expression, until we reach a size/length attribute of an array or a known collection class from the Java Collection Library. To make sure that the collection size is comparable with the loop count, we consider only two types of data dependencies: (1) direct assignment (e.g., `a = b;`), and (2) addition or subtraction expression with one operand as constant (e.g., `a = b.size() - 1`).

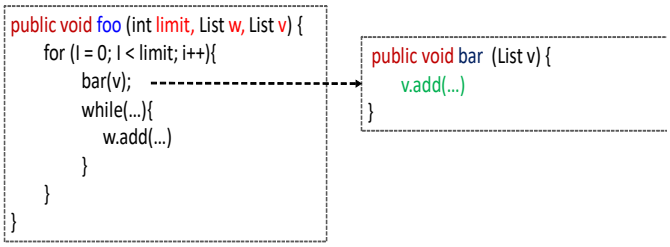

Figure 4.5: Code Sample of Operation Dependency

To identify operation dependencies, for each loop L, we check its body for element-addition and element-removal operations on collection variables. For any other method invocations in the loop, we recursively go into the body of each invoked method to further look for such operations. However, we do not consider nested-loop blocks in L or a method invoked from L, because such blocks are dependent on their direct enclosing loop. For example, in Figure 4.5, collection variable v is operation-dependent on Loop$_A$, but w is not (it is operation-dependent on Loop$_B$).

After identifying these two types of dependency relations, we further apply points-to analysis [73] to identify alias relations among collection variables. Note that in our analyses we consider only variables of known collection classes from the Java Collection Library. User-defined collections are also common. However, since we use inter-procedural analysis for identifying both types of dependencies, as long as the user-defined collections extend or wrap Java-Collection classes from the Java Collection Library, we are able to handle these user-defined collections by building dependencies directly on the Java-Collection variables inside them. Also note that we identify all dependencies and alias relations on the base version and record the results so that we need to re-analyze only the revised/added methods when a code commit comes.

Iteration-Count Inference

With the dependencies identified among collections and loops, when a new code commit comes, we use Algorithm 1 to infer the iteration count of new loops and update the iteration count of existing affected loops.

In the algorithm, we use a work queue to iteratively update sizes of collection variables and loop-iteration counts (stored in $lCount$), and we use the combined map of $MapI$ and $MapO$ to

Algorithm 1: Iteration Count Inference

Require:
$MapO$ is a map from collection variables to loops
$MapI$ is a map from loops to collection variables
$lCount$ is a map from loops to iteration counts
Ensure:
updated $lCount$

```
1:  Q ← MapI.keys()
2:  Map ← MapO ⋃ MapI
3:  while Q ≠ ∅ do
4:      top ← Q.pop()
5:      for all val ∈ Map.get(top) do
6:          if val ∉ lCount.keys() then
7:              lCount.add(val, lCount.get(top)
8:              Q.add(val)
9:          else if val is a collection variable then
10:             lCount.set(val, lCount.get(val) + lCount.get(top)
11:             Q.add(val)
12:         end if
13:     end for
14: end while
15: lCount.removeAll(MapO.keys())
```

transit between collections and loops. In particular, as shown in Lines 6-11, we update the iteration count of each loop at most once, to avoid infinite update process caused by cyclic dependency (the more in-depth reason is that we use numbers to represent iteration counts, which are not in a bounded domain). In the end, we remove collection variables from $lCount$ to retain only the loops in the map.

4.3.5 Test Case Prioritization

Once our approach estimates the performance impact of the code commit on each test case, we can rank test cases according to their relative performance impact. We use the main method as the root for system tests and each test method as the root for unit tests. We consider both positive and negative effect on execute time as it is often also important for developers to understand whether and where their commit is able to enhance the software performance.

4.4 Evaluation

For our evaluation, we implement PerfRanker based on Soot for static analysis and Java Agent for profiling the base version.

4.4.1 Evaluation Subjects

We apply PerfRanker on two popular open source projects: Xalan [3] and Apache Commons Math [1]. Specifically, Xalan is an XSLT processor, and Apache Commons Math is a library for mathematical operations. We choose these two projects to cover both data formatting and mathematical computations, which are two representative time-consuming components in modern software. Xalan is equipped with a performance test suite of 64 test cases. Since Apache Commons Math is not equipped with a performance test suite, we leverage its unit test cases as performance test cases.

Version Selection. For Apache Commons Math, we use its version on Jan 1, 2013 as its base version. For Xalan, since there are very few code commits after 2013, we use version 2.7.0 as its base version, as 2.7.0 is the first Xalan version compatible with Java 6 and higher. For both software projects, we collect all code commits from the base version until Mar 17, 2016, the time when we started collecting data for our work. From all code commits, we remove those that do not change source files and those that do not involve semantic changes (e.g., renaming variables), as developers can easily determine that those commits will not affect software performance. Furthermore, we choose as our code-commit set the top 15 code commits whose changed code portions are covered by most test cases, where test prioritization is most needed.

In Table 4.1, we present some statistics of the studied subjects and versions. The table shows that either project has more than 300K lines of code. Furthermore, there are hundreds of code commits and changed files between the base version and our selected code commits. In our evaluation, we do not update the base version, so the overhead of profiling the base version is low compared with the number of code commits under study. More details about our evaluation subjects can be found on our project website [8].

Table 4.1: Evaluation Subjects

Subject	Xalan	Apache Commons Math
Base Ver.	2_7_0	Jan 1st, 2013
Size (LOC) of Base Ver.	413,534	398,171
# Commits Since Base Ver.	354	1,321
# Changed Files	1,206	1,613
Last Commit Date	Aug 11, 2015	Mar 17, 2016

4.4.2 Evaluation Setup

To determine performance regressions as the ground truth of performance changes for all test cases and code commits, we execute the test cases for 5,000 times on the base version and each code commit under study. Furthermore, we execute the base version with our Java Agent to record the dynamic call graph and the execution time of each method, as well as the iteration number of each loop. To record average execution time of methods defined in the JDK library, we execute the Dacapo benchmark 9.12 [21] with profiling (we remove Xalan from the benchmark to avoid bias). All the executions are conducted on a Dell x630 PowerEdge Server with 32 cores and 256GB memory, and the server is used exclusively for our evaluation to avoid noises.

4.4.3 Evaluation Metrics

To the best of our knowledge, our work is the first on prioritizing performance test cases for code commits, and we propose a set of metrics to evaluate the quality of different rankings. In our evaluation, we consider three ranking metrics: Average Percent of Fault-Detection on Performance (*APFD-P*), normalized Discounted Cumulative Gain (*nDCG*), and *Top-N Percentile*.

APFD-P. *APFD* [80] is a commonly used metric for assessing a test sequence produced by test-case prioritization. If the test-suite size is N, the total number of faults detected by the test suite is T, and the number of faults detected by the first x test cases in the test sequence is $detected(x)$, then the *APFD* of the test sequence can be defined in Formula 4.5:

$$APFD = \frac{\sum_{x=1}^{N} \frac{detected(x)}{T}}{N} * 100\% \tag{4.5}$$

Unlike functional bugs where a test case either passes or fails, performance regressions are not binary but continuous. Performance downgrades of 20% and 50% are both regressions, with different severity. Therefore, instead of counting detected faults to attain the value of $detected(x)$, we replace the value of $detected(x)$ with the accumulated performance change. We define the *performance change* of the i^{th} test case in the test sequence (denoted as $change(i)$) in Formula 4.6 as below, in which $exe(i)$ is the execution time of the i^{th} test case in the current version, and $exe(i_{base})$ is the execution time of the i^{th} test case in the base version.

$$change(i) = \frac{|exe(i) - exe(i_{base})|}{exe(i_{base})} \tag{4.6}$$

Then, we define *APFD-P* the same as Formula 4.5, except that $detected(x)$ is defined as the accumulated performance change, as shown in Formula 4.7, and T is the sum of performance changes on all test cases. Actually, with such a definition, *APFD-P* can be viewed as *APFD* where all test cases reveal faults, and these faults are weighted by performance changes.

$$detected(x) = \sum_{i=1}^{x} change(i) \tag{4.7}$$

As an illustrative example, consider 3 tests t_1, t_2, and t_3 with 10%, 20%, and 30% performance downgrades, respectively. The best ranking is t_3, t_2, t_1; the total performance impact is $10\% + 20\% + 30\% = 60\%$; and the covered performance impact after each test is $30\%/60\% = 50\%$, $(30\% + 20\%)/60\% = 83\%$, and $(30\% + 20\% + 10\%)/60\% = 100\%$. The P-APFD is thus (50% + 83% + 100%)/3 = 78%.

nDCG. nDCG [11] is a metric of ranking widely used in information retrieval. The basic idea is to calculate the relative score a given ranking with an ideal ranking, and the score of an arbitrary ranking is defined below, where $change(i)$ is defined in Formula 4.6.

$$DCG(seq) = change(1) + \sum_{i=2}^{N} \frac{change(i)}{log_2(i)} \tag{4.8}$$

Top-N Percentile. The *APFD-P* and *nDCG* defined earlier are adapted versions of widely used metrics, and can be used to compare different prioritization approaches. However, they are

not sufficiently intuitive to help understand how much developers can benefit from an approach. Therefore, in our evaluation, we also measure how high percentage of top-ranked test cases in a test sequence need to be executed to cover the test cases with top N performance impacts (we use 1 and 3 for N). For example, if the test cases with top 1, 2, and 3 performance impacts are ranked in the 2^{nd}, 9^{th}, and 5^{th} positions in a test sequence with length 100, then the top 1, 2, and 3 percentiles are 2%, 9%, and 9%, respectively.

4.4.4 Baseline Approaches Under Comparison

Although we are not aware of approaches specifically designed for prioritizing performance test cases in regression testing, it is possible to adapt existing approaches for performance test prioritization. In our evaluation, we compare our approach with three baseline approaches: Change-Aware Random, Change-Aware Coverage, and Change-Aware Loop Coverage.

Specifically, in all baseline approaches, we apply change-impact analysis to rule out the test cases that do not cover any revised methods. Since our performance-impact analysis includes basic change-impact analysis, for fair comparison, we apply this change-impact analysis in all baseline approaches. Note that we gathered coverage information from the base version, and we use the same technique as in our approach when selecting the code commits affecting most test cases in the three baseline approaches.

After selecting the relevant test cases, the *Change-Aware Random (CAR)* approach simply ranks the test cases in random order[1]. The *Change-Aware Coverage (CAC)* approach applies coverage-based test prioritization [111] on the covered methods with the additional strategy [136], being a state-of-the-art approach in defect-oriented test prioritization. The basic idea is to first select the test case with the highest coverage, and iteratively select the test case that covers the most not-covered code portions as the next test case. In our evaluation, we use method coverage as the criterion, being consistent with the granularity of our performance model. The *Change-Aware Loop Coverage (CALC)* approach is the same as CAC, except for using coverage of loops instead

[1]To acquire more stable results, we use the average result of 100 random ordered test sequences as the result for CAR.

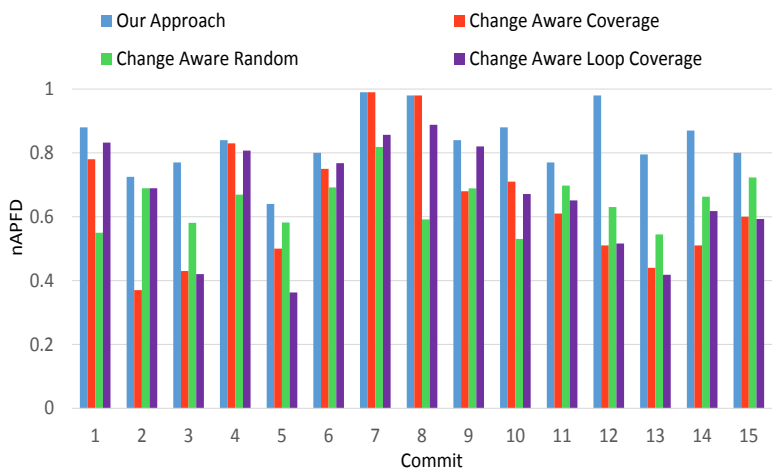

Figure 4.6: *APFD-P* Comparison on Apache Commons Math

of methods as the criterion.

4.4.5 Quantitative Evaluation

In our quantitative evaluation, we compare our approach and the three baseline approaches on all three metrics.

APFD-P Metric

Figures 4.6 and 4.7 show the comparison results between our approach and three baseline approaches on the *APFD-P* metric. In the two figures and all the following figures, the X axis lists all the code commits studied chronologically, and the Y axis shows the *APFD-P* value (or *nDCG* value). We use different colors to represent different approaches consistently for all figures according to the legend in Figure 4.6.

Figures 4.6 and 4.7 show that our approach is able to achieve over 80% *APFD-P* value in most code commits affecting performance, and outperforms or rivals all baseline approaches in all code commits affecting performance from both subject projects. Specifically, for Apache Commons Math, our approach achieves an average *APFD-P* value of 83.7%, compared with 64.3% by CAR, 64.6% by CAC, and 66.1% by CALC. For Xalan, our approach achieves an average *APFD-P* value of 83.5%, compared with 65.8% by CAR, 63.6% by CAC, and 59.8% by CALC. Therefore, the

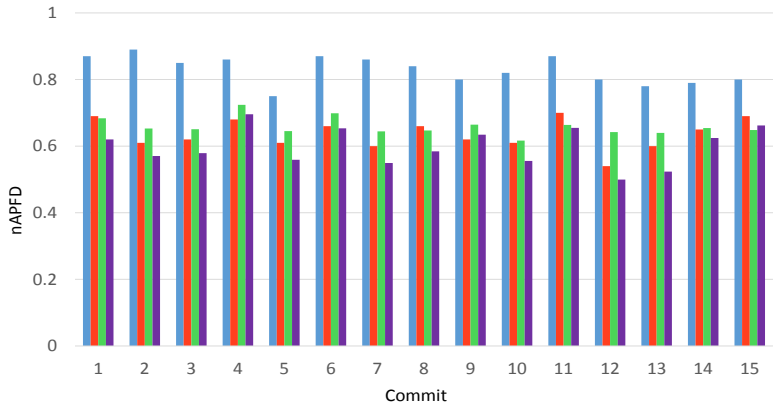

Figure 4.7: *APFD-P* Comparison on Xalan

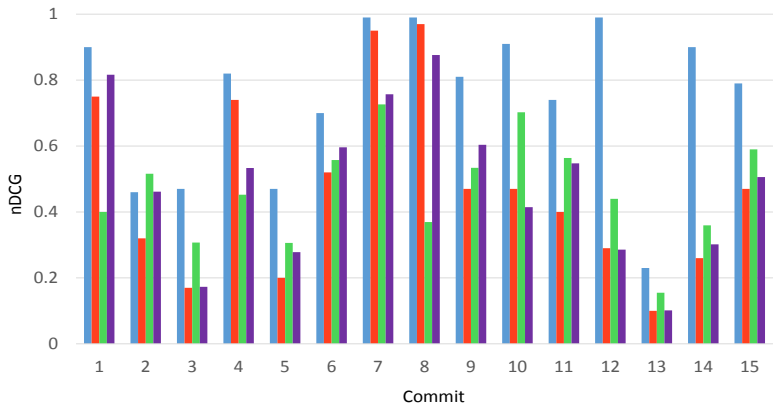

Figure 4.8: *nDCG* Comparison on Apache Commons Math

improvement on the average *APFD-P* is at least 17 percentage points, compared with baseline approaches on both projects. Furthermore, we do not observe significant effectiveness downgrade in the later versions, indicating that one base version can be used for a relatively long time.

nDCG Metrics

Similarly, Figures 4.8 and 4.9 show the comparison results between our approach and the three baseline approaches on the *nDCG* metric.

The figures show that our approach outperforms or rivals baseline approaches on *nDCG* in almost all code commits from both subject projects. Specifically, for Apache Commons Math, our approach achieves an average *nDCG* value of 74.5%, compared with 47.2% by CAR, 46.5% by CAC, and 48.4% by CALC. For Xalan, our approach achieves an average *nDCG* value of 71.7%, compared with 43.0% by CAR, 41.4% by CAC, and 37.4% by CALC. Therefore, the improvement

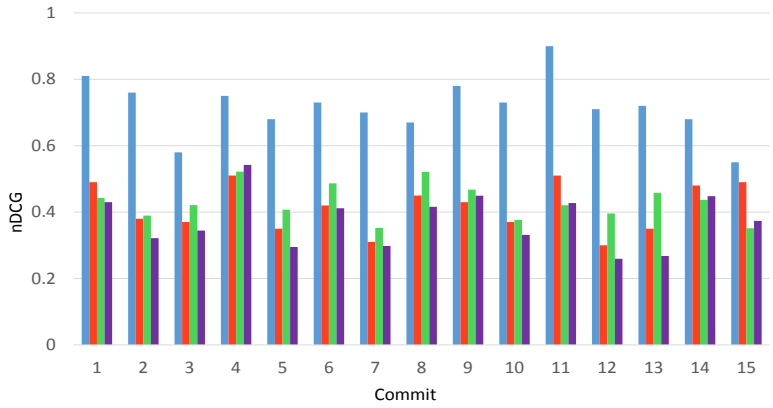

Figure 4.9: *nDCG* Comparison on Xalan

on the average *nDCG* is over 26 percentage points on both projects. We also observe that there is one code commit from Xalan, in which our approach performs slightly worse than CAR on *nDCG*. In Section 4.4.6, we further discuss the details of this code commit in Listing 4.4.

Finally, we observe that compared with *APFD-P*, the *nDCG* values are generally lower and vary more significantly from code commit to code commit. The reason is that an *nDCG* value is more sensitive to the rank of test cases with the highest performance impacts. For example, consider a test sequence with 100 test cases and only 1 test case has performance impact, and the performance impact value is 100%. When this test case is ranked top, both *APFD-P* and *nDCG* are 1.0. However, if this test case is ranked 25^{th} in the sequence, the *APFD-P* value is still as high as 75%, but the *nDCG* value becomes $1/log_2(25)$, which is less than 25%. This result is reasonable because in information retrieval (where *nDCG* was first proposed), ranking the most relevant result at the 25^{th} position is very bad, but in test prioritization (where *APFD* was first proposed), ranking the test case at the 25^{th} position is not so bad, because only 25% test cases need to be executed to execute the test case. Therefore, which of *APFD-P* and *nDCG* is a better metric may depend on whether developers are interested in only a few most severely affected test cases, or a larger number of test cases whose performance is affected.

Top-N Percentile

While *APFD-P* and *nDCG* are normalized quantitative metrics for our problem, they are not sufficiently intuitive for understanding the direct benefit of our approach on developers. Therefore,

Table 4.2: Top-N Percentile of Apache Commons Math

C #	T #	Top 1				Top 3			
		Our	CAC	CALC	CAR	Our	CAC	CALC	CAR
1	55	2%	3%	**1%**	49%	**6%**	43	27%	74%
2	60	40%	61%	**38%**	49%	**51%**	61%	93%	74%
3	152	**2%**	84%	79%	49%	**28%**	88%	88%	74%
4	97	**3%**	18%	87%	49%	**5%**	18%	87%	74%
5	130	**4%**	29%	43%	49%	**4%**	96%	97%	74%
6	15	**13%**	40%	33%	46%	66%	**40%**	100%	73%
7	18	**5%**	**5%**	88%	47%	**15%**	40%	88%	74%
8	10	**10%**	**10%**	40%	45%	**60%**	**60%**	70%	70%
9	12	**8%**	66%	75%	45%	**50%**	66%	75%	72%
10	12	**8%**	50%	83%	45%	83%	**50%**	100%	72%
11	36	**5%**	94%	38%	48%	66%	94%	**58%**	74%
12	13	**7%**	76%	38%	45%	76%	84%	100%	**72%**
13	711	**7%**	81%	84%	49%	**7%**	81%	84%	74%
14	39	**2%**	84%	25%	48%	**12%**	84%	79%	74%
15	34	**11%**	52%	17%	48%	**26%**	58%	**26%**	74%
Avg		**8%**	50%	51%	48%	**37%**	65%	78%	73%

Table 4.3: Top-N Percentile of Xalan

C #	T #	Top 1				Top 3			
		Our	CAC	CALC	CAR	Our	CAC	CALC	CAR
1	63	20%	46%	**14%**	49%	**36%**	60%	50%	74%
2	63	**17%**	85%	22%	49%	**17%**	85%	80%	74%
3	63	**7%**	85%	50%	49%	**38%**	85%	66%	74%
4	63	20%	80%	**7%**	49%	**20%**	85%	73%	74%
5	63	**6%**	85%	100%	49%	**53%**	85%	100%	74%
6	63	**11%**	63%	31%	49%	**26%**	76%	77%	74%
7	63	**11%**	46%	12%	49%	**15%**	85%	93%	74%
8	63	**9%**	35%	36%	49%	**20%**	82%	95%	74%
9	58	**3%**	96%	5%	49%	**25%**	96%	98%	74%
10	63	30%	47%	**17%**	49%	**31%**	85%	84%	74%
11	63	**1%**	31%	12%	49%	**7%**	62%	74%	74%
12	63	**23%**	85%	88%	49%	**34%**	90%	88%	74%
13	63	**12%**	46%	82%	49%	**53%**	65%	82%	74%
14	58	34%	34%	**8%**	49%	43%	**37%**	72%	74%
15	63	36%	**4%**	12%	49%	**36%**	79%	61%	74%
Avg		**16%**	57%	34%	49%	**30%**	77%	79%	74%

we further measure how many test cases developers need to consider if they want to cover the top 3 most affected test cases. Tables 4.2 and 4.3 show the results. Columns 1 and 2 present the code commit number and the total number of test cases affected by the code commit, respectively. Columns 3-6 and 7-10 present the top proportion of ranked test cases required to cover top 1 and 3 most-performance-affected test cases[2].

Tables 4.2 and 4.3 show that on average our approach is able to cover Top 1 and 3 most-performance-affected test cases within top 8%, 21%, and 37% of ranked test cases in Apache Commons Math, and 16%, 22%, and 30% of test cases in Xalan. The improvements over the baselines approaches are at least 33%, 37%, and 28%. Furthermore, on top 1 coverage, our approach

[2]The results for top 2 show similar trends and are available on the project website [8]

outperforms or rivals the best baseline approach on 13 code commits from Apache Commons Math, and 10 code commits from Xalan. On top 3 coverage, our approach achieves the highest percentage on 11 code commits from Apache Commons Math, and 14 code commits from Xalan.

Overhead and Performance

In test prioritization, it is important to make sure that the time spent on prioritization is much smaller than the execution time of performance test cases. To confirm indeed that is the case, we record the overhead and execution time of our analysis. Our profiling of the base version has a 1.92 times overhead on Apache Commons Math, and 5.18 times overhead on Xalan. The static analysis per test case takes 29.90 seconds on Apache Commons Maths, and 34.35 seconds on Xalan. Finally, for Apache Commons Math, the average, minimal, and maximal time for analyzing a code commit are 45.35 seconds, 4.23 seconds, and 262.30 seconds, respectively, while for Xalan, the average, minimal, and maximal time for analyzing a code commit are 9 seconds, 3.36 seconds, and 21.80 seconds, respectively. In contrast, it takes averagely 52 (3) minutes to execute test suite of Apache Commons Math (Xalan) for 173 (47) times to achieve an expectation of equal to or less than 10% execution-time variance.

4.4.6 Successful and Challenging Examples

In this section, with representative examples of code commits, we explain why our approach performs well on some code commits but not so well on some others.

Successful Example 1. Listing 4.2 shows the simplified code change of code commit hash d074054... of Xalan. On this code commit, our approach is able to improve the *APFD-P* and *nDCG* values by at least 13.53 and 31.22, respectively, compared with the three baseline approaches. In this example, several statements are added inside a loop. Our approach can accurately estimate the performance change because (1) Line 2 in the code is an existing loop and from the profile database for the base version we can find out exactly how many times it is executed, and (2) the added method invocations are combinations of existing method invocations whose execution time

is already recorded for the base version.

```
1    int nAttrs = m_avts.size();
2    for (int i = (nAttrs - 1); i >= 0; i--){
3        ...
4  +   AVT avt = (AVT) m_avts.get(i);
5  +   avt.fixupVariables(vnames, cstate.getGlobalsSize());
6        ...
7    }
```

Listing 4.2: Change Inside a Loop

Successful Example 2. Listing 4.3 shows the simplified code change of code commit hash 64ec535... of Xalan. On this code commit, our approach is able to improve the *APFD-P* and *nDCG* values by at least 17.14 and 24.33, respectively, compared with the baseline approaches. In this example, the loop at Line 4 depends on the collection variable `m_prefixMappings` at Line 1 whose size can be inferred from the recorded number of iterations of existing loops. In this case, our approach can accurately estimate the iteration count of this new loop and estimate the overall performance impact based on the estimated iteration count.

```
1  + int nDecls = m_prefixMappings.size();
2  +
3  + for (int i = 0; i < nDecls; i += 2){
4  +   prefix = (String) m_prefixMappings.elementAt(i);
5  +   ...
6  + }
```

Listing 4.3: A Newly Added Loop Correlating to an Existing Collection

Challenging Example 1. Listing 4.4 shows the simplified code change of code commit hash 90e428d... of Apache Commons Math. On this code commit, with respect to the *nDCG* metric, our approach performs better than the CAC baseline approach but slightly worse than the CAR baseline approach. In the example, at Line 2, an invocation to method `checkParameters()` is added, and the method may throw an exception. In this example, the execution time of `checkParameters()` can be easily estimated with our performance model. However, if the exception is thrown, the rest of the method will not be executed. Although we are able to estimate the execution time of the method's remaining part, it is impossible to estimate the probability of throwing the exception, as `checkParameters()` is a newly added method without any profile information. In such cases, if the probability of throwing the exception is higher in some test cases, the reduction of execution

time due to the exception will be the dominating factor and result in inaccuracy in our prioritization.

```
1    protected PointVectorValuePair doOptimize(){
2  +   checkParameters();
3        ...
4    }
5  + private void checkParameters() {
6  +   ...
7  +   throw new MathUnsupportedOperationException;
8  + }
```

Listing 4.4: Return or Throw Exception at Beginning

Challenging Example 2. There are also cases where developers added a loop that is not relevant to any existing collection variables. As one of such examples, Listing 4.5 shows the simplified code change of code commit hash a51119c... of Apache Commons Math. On this code commit, the improvement of our approach over the best result of the three baseline approaches is only 0.8 for *APFD-P* and 6.0 for *nDCG*. In the example, Line 2 introduces a new loop that does not correlate to any existing collection or array. In such a case, our approach cannot determine the iteration count of this loop and the depth of recursion at Line 8. To still provide prioritization results, our approach uses the average iteration counts of all known loops to estimate the iteration count of this new loop and always estimates the recursion depth to be 1. However, our prioritization result becomes less precise due to such coarse approximation.

```
1    public long nextLong(final long lower,  final  long upper){
2  +   while (true) {
3          ...
4  +       if (r >= lower && r <= upper) {
5  +         return r;
6  +       }
7  +   }
8  +   return lower + nextLong(getRan(), max);
9    }
```

Listing 4.5: New Loop with No Correlation

4.4.7 Threats to Validity

Major threats to internal validity are potential faults in the implementation of our approach and baseline approaches, potential errors in computing evaluation results of various metrics, and the various factors affecting the recorded execution time for the test cases. To reduce such threats,

we carefully implement and inspect all the programs, and execute the test cases for 5,000 times to reduce random noises in execution time. Major threats to external validity are that our evaluation results may be specific to the code commits and subjects studied. To reduce the threats, we evaluate our approach on both data processing/formatting software and mathematical computing software, and both a unit test suite and a performance test suite.

4.5 Discussion

Handling Recursions. Recursions and loops are two major ways to execute a piece of code iteratively. Our approach constructs dependency relationships between loops and collection variables to estimate the iteration counts of loops. For recursion cycles existing in the base version, we simply update execution-time changes along the cycle only once, and multiply the execution-time change by averaging the invocation depths, attained via dividing the total execution frequency of all methods in the cycle by the product of invocation-frequency sum of these methods from outside the cycle and the accumulated recursive invocations inside the cycle (multiplying invocation frequencies along the cycle once). As an example, consider a cyclic call graph: $X \rightarrow A$, $Y \rightarrow B$, $A \rightarrow B$, and $B \rightarrow A$. When $t(B)$ is changed, the impact is propagated to A as $t(A) = t(B) * f_{AB}$. The propagation then stops to break the cycle. After that, impact on X becomes $f_{XA} * depth * t(A)$ where $depth$ is the cycle's number of execution iterations, estimated as below:

$$\frac{total(A) + total(B)}{(f_{XA} + f_{YB}) * f_{AB} * f_{BA}} \tag{4.9}$$

where $total(F)$ is the total frequency of method F in profile, and the divisor is the product of all calling frequencies from outside and inside of a cycle. For newly added recursions, our approach currently does not support estimation of the invocation depth, and simply assumes the invocation depth to be 1. In future work, we plan to develop analysis to estimate the termination condition of newly added recursions and relate invocation depths to collection variables.

Approximation in Performance Estimation. Since our approach is not able to make any assumption on the given code commit, we have to make coarse approximations for the parameters

of our performance model. For example, we ignore non-method-call instructions in revised methods, assume all newly added recursions to have invocation depth 1, and use the average recorded execution time of existing methods and JDK library methods to estimate their execution time in the new version. More advanced analysis can result in more accurate execution-time estimation, yet with higher overhead in the prioritization process, so future investigation is needed for the best trade-off.

Supporting Multi-threaded Programs. Multi-threaded programs are widely used for high-performance systems. In multi-threaded programs, methods and statement blocks can be executed concurrently, and thus our performance model can be inaccurate because the product of invocation frequency and average execution time of a method is no longer the total execution time. To address concurrent execution, we need to analyze the base version to find out the methods that can be executed concurrently. Then, we can give such methods a penalizing coefficient to reflect the extent of concurrency. We plan to explore this direction in future work.

Selection of Base Versions. The overhead of our approach largely depends on the required number of base versions. In our evaluation, we use one base version to estimate the subsequent code commits ranging over more than 3 years and 300 code commits. The evaluation results do not show significant effectiveness downgrade as time goes by. One potential reason is that, both software projects in our evaluation are in their stable phase and the code commits are less likely to interfere with each other.

4.6 Related Work

Performance Testing and Faults. Previous work focuses on generating performance test infrastructures and test cases, such as automated performance benchmarking [61], model-based performance testing framework for workloads [15], using genetic algorithms to expose performance regressions [79], learning-based performance testing [49], symbolic-execution-based load-test generation [137], probabilistic symbolic execution [26], and profiling-based test generation to reach performance bottlenecks [78]. Pradel et al. [104] propose an approach to support generation

of multi-threaded tests based on single-threaded tests. Kwon et al. [67] propose an approach to predict execution time of a given input for Android apps. Bound analyses [51] try to statically estimate the upper bound of loop iterations regarding input sizes, but they cannot be directly applied as the size of collection variables under a certain test can be difficult to determine. Most recently, Padhye and Sen [97] propose an approach to identify collection traversals in program code; such approach has the potential to be used for execution-time prediction. In contrast to such previous work, our approach focuses on prioritizing existing performance test cases. The most related work in this direction is done by Huang et al. [56], whose differences with our approach are elaborated in Section 5.1.

Another related area is research on performance faults, including studies on performance faults [59, 134], static performance-fault detection [60, 62, 95, 132], debugging of known performance faults [10, 53, 71, 114], automatic patches of performance faults [94], and analysis of performance-testing results [40, 41].

Test Prioritization and Impact Analysis. Test prioritization is a well explored area in regression testing to reduce test cost [20, 55, 138] or to detect functional faults earlier [36, 63, 74]. Mocking [90] is another approach to reduce test cost, but it does not work for performance testing as mocked methods do not have normal execution time. Another related area is test selection or reduction [27, 54, 110] which sets a threshold or other criteria to select/remove part of the test cases. Most of the proposed efforts are based on some coverage criterion for test cases, and/or impact analysis of code commits. The impact analysis falls into three categories: static change impact analysis [13, 119, 123], dynamic impact analysis [12, 70, 96], and version-history-based impact analysis [87, 115, 139]. Our approach leverages a similar strategy to rank performance tests according to the change impact on them. However, we propose specific techniques to estimate performance impacts, such as collection-loop correlation and performance impact analysis.

4.7 Conclusion

In this paper, we present a novel approach to prioritizing performance test cases according to a code commit's performance impact on them. With our approach, developers can execute most-affected test cases earlier and for more times to confirm a performance regression. Our evaluation results show that our approach is able to achieve large improvement over three baseline approaches, and to cover top 3 most-performance-affected test cases within 37% and 30% test cases on Apache Commons Math and Xalan, respectively.

Chapter 5: BEYOND API SIGNATURES: BEHAVIORAL BACKWARD INCOMPATIBILITIES OF JAVA SOFTWARE LIBRARIES

5.1 Introduction

Nowadays, as software products become larger and more complicated, software libraries have become a necessary part of almost any software. Since software libraries and their client software are typically maintained by different developers, the asynchronous evolution of software libraries and client software may result in incompatibilities. To avoid incompatibilities, for decades, "backward compatibility" has been a well known requirement in the evolution of software libraries. Each API method in an existing version of software library should exactly maintain its behavior in the following versions.

However, in reality, full backward compatibility is seldom achieved, and the resultant incompatibilities have been known to affect software users as well as the success of both software libraries and client software. For example, criticism on Windows Vista can be largely ascribed to its backward incompatibility with Windows XP [7]. In October 2014, the automatic system update to Android 5.0 caused a complete dysfunction of SogouInput (the top Android app for Chinese input on mobile phones, with more than 200 million users), which is not patched until 3 days later [9]. Also, a recent study [75] has shown that the usage of unstable Android APIs is an important factor affecting the success of Android apps. Therefore, we believe that it is necessary to conduct thorough studies to understand the status and reasons of backward incompatibilities, and the result of such studies may lead to more advanced techniques to support detection, documentation, and resolution of backward incompatibilities.

There have been many existing empirical studies [85, 106] on the stability of software libraries, and extensive research efforts on library migration [30, 127]. However, these studies mainly focus on the signature incompatibilities (i.e., added, revised, and removed API methods) between consecutive versions of software libraries. Although API signature changes form an important cat-

egory of backward incompatibilities, they do not describe the whole picture. Even if the signature of an API method remains identical in a new version, it is possible that its behavior changes (i.e., it generates different output or side effect when certain input are fed in). We refer to such behavioral changes as *Behavioral Backward Incompatibilities* (BBIs). Actually, since signature incompatibilities can be detected by compilers, although they may cause extra efforts in library migration, they are less likely to cause real-world bugs, and thus less harmful compared to BBIs.

To acquire a deeper and more complete understanding of BBIs in real world software development, in this paper, we present an experimental study on 68 consecutive version pairs from 15 popular Java software libraries. For each pair, we performed cross-version testing to test the new version of software library with the test code of the old version, and manually inspected and categorized the test errors / failures. We further collected and manually inspected 126 real-world bug reports related to BBIs in these libraries. We chose Java software libraries as our subjects because Java software typically extensively relies on libraries, and most popular Java software libraries are open source with test code available.

The three main contributions this paper makes include:

- A large scale experimental study on BBIs from 15 popular Java libraries and 68 version pairs, grouping 1094 detected test errors / failures to BBIs, and categorizing BBIs by incompatible behaviors, invocation conditions, and reasons.
- A qualitative study of 126 real world bugs caused by BBIs, including the categorization of their root-cause BBIs, and their resolutions.
- A data set including 296 BBIs detected in cross-version testing, and 126 real world bugs, serving as a basis for future research in this area.

5.2 Research Scope

In our study, we try to answer the five research questions as follows.

- **RQ1:** Are BBIs prevalent between consecutive version pairs of Java software libraries?

- **RQ2:** Are most BBIs distribute in the major version upgrades, so that minor version updates are safer?
- **RQ3:** What are the characteristics of BBIs and why library developers bring them in?
- **RQ4:** How are BBIs detected in regression testing compare with BBIs causing real-world bugs?
- **RQ5:** How are the BBI-related bugs fixed in software practice?

With the answers of these questions, we expect to understand: (1) whether BBIs are prevalent in the Java software libraries, and a problem that software developers need to face frequently, (2) whether BBIs are distributed most in major version upgrades, which are supposed to have some software interface changes, (3) whether it is possible to classify BBIs into several categories by their characteristics and reasons brought in, so that they can be avoided or detection and resolution techniques can be developed accordingly, (4) whether there are mismatches on the categories of BBIs being most detected and the categories of BBIs mostly likely to cause bugs, and (5) whether BBIs are fixed by library or client developers, and whether there exists certain fixing patterns on BBI related bugs.

5.3 Cross-Version Testing Study

In this section, we introduce our experimental study[1] on consecutive versions of Java libraries with cross-version testing.

5.3.1 Study Setup

Subject Libraries

In our experimental study, we used 15 popular Java libraries as our subjects as shown in Column 1 of Table 5.1. Specifically, we include in our subjects the two most widely used Java libraries:

[1]Our data for this study and the following bug study are both available at `https://sites.google.com/site/incompemp2017/`.

OpenJDK and the Android framework. Actually, since these two libraries have corresponding runtime platforms (i.e., JVM and Android OS), their BBIs are more likely to cause runtime errors, because the old version of client software may be executed in JVM or Android without recompile, and thus may result in runtime errors. Our subjects also include 7 libraries from Apache, and 6 other third-party libraries. All these libraries are from different domains and are the most popular software libraries in their domains according to a statistics [90] [91] of class imports among top 5,000 Java software projects in Github.

Selection of Version Pairs

Software developers use different levels of versions to mark different granularity of milestones in software evolution. In our study, we first rule out the alpha and beta versions which are typically immature versions and are not widely used by client software developers. Then, we also need to differentiate major versions and minor versions. According to Semantic Versioning [2] [107], backward-incompatible API changes can be allowed in major versions (e.g., Java 6), but not minor versions (e.g., Java6u32). Also, major versions are typically developed in separate branches, while minor versions just corresponds to certain commits in the trunk or a branch.

To acquire a full picture, in our study, we study backward incompatibilities both between two consecutive major versions and within a major version. If a major version has more than two minor versions, we choose the first minor version and the last minor version to form an inner-major-version version pair. For example, ElasticSearch has four minor versions (1.0.0 through 1.0.3) for major version 1.0, and three minor versions (1.1.0 through 1.1.2) for major version 1.1. So, in our study, we choose four versions (1.0.0, 1.0.3, 1.1.0, 1.1.2), and form 1 major version pairs (1.0.3 to 1.1.0) and two minor version pairs (1.0.0 to 1.0.3, 1.1.0 to 1.1.2). We combine minor versions within a major version because they typically contain very small amount of changes (e.g., fixing a bug), and may be inverted in the later updates if bringing in bugs or BBIs. So the combination will remove temporary BBIs (brought in and fixed with in a major version) which may not affect client developers much. We also ruled out the versions that raise compilation errors, or

Table 5.1: Basic Information of Studied Subjects and Versions

Subject	St. V.	End V.	# V.	St. Time	End Time
OpenJDK	7b157	8b13	2	2011-7	2014-3
Android	4.3.1	5.0.1	2	2013-10	2014-12
log4j	2.0.0	2.1	2	2014-7	2014-10
maven	3.0.0	3.2.5	4	2010-10	2014-12
bukkit	1.2.3	1.7.2	6	2011-12	2013-12
beanutils	1.9.0	1.9.2	1	2008-9	2013-12
codec	1.6	1.7	1	2011-11	2012-9
fileupload	1.2.0	1.3.1	3	2007-2	2014-2
commons-io	2.0	2.4	4	2007-7	2012-4
ela. Search	1.0.3	1.3.9	7	2014-4	2015-2
http-core	4.0.1	4.3.3	6	2009-2	2014-2
jodatime	2.0	2.7	7	2011-5	2015-1
jsoup	1.1.1	1.7.3	10	2010-6	2013-11
neo4j	1.8.3	2.0.3	5	2012-11	2015-2
snakeyaml	1.3	1.11	8	2009-7	2012-9

unit test failures / errors[2]. Finally, we use only versions up to Jan. 2015, so that their status (e.g., documentation content) is relatively stable. Details of our selected versions as presented in Column 2-4 of Table 5.1 (Column 5-6 present the release time of the first and last version of the subject used in our study).

Detection of BBIs

Not all test cases in the old version compile with the source code of the new version (typically because they suffer from signature incompatibilities). Therefore, to detect behavioral incompatibilities, for each version pair, we automatically recompiled the test code of previous version with the source code of the new version, and iteratively remove the test cases that do not compile with the new version of source code, until all test cases can be compiled successfully. Finally, we executed all the remaining test cases and collected test failures and errors. Specifically, test compilation errors appear in 37 versions from 12 subjects (except for BeanUtils, FileUpload, and Codec), and

[2]For JDK and Android, we used the versions despite test failures and errors because some test cases require hardware support that we do not have. For JDK and Android versions, we ignore the test cases that fail on their own version.

in total 2,590 of 57,208 test cases (4.5%) are removed due to compilations errors.

Since one BBI may causes multiple test failures and errors, we further manually inspected these test failures and errors and grouped them into BBIs. For each test error/failure, we extracted the error messages, the version diff of the failed test code, and the version diff of the test class and the source class being tested. Then, we categorize multiple test errors/failures as one BBI if the test errors/failures are caused by the same API method and result in the same error message, exception, or wrong value of the same output/side effect. Note that, among 296 backward incompatibilities, library developers have revised test cases (e.g., changing test oracles or ways of invocation) for 267 incompatibilities, and deleted test cases for the other 11. For the rest 18 incompatibilities, change was made in other places of the test suite such as revising the setting up code or upgrading referred libraries. We found that revised test cases can help a lot in understanding the behaviors of BBIs. This categorization was done by first 2 authors separately, with the third author as a judge for conflicts.

5.3.2 BBIs in Popular Libraries

To answer **RQ1**, we present the detected test failures / errors from software-library consecutive version pairs in Table 5.2. The first column of the table presents the subject name. Columns 2-3 present the total number of test failures detected in all version pairs of a specific subject (abbreviated as $T.$), and the number of versions where test failures are detected (denoted as I) divided by all version pairs of the subject (denoted as A). Columns 4-5 and 6-7 present similar data for test errors and BBIs (after grouping). Note that a test failure is raised when an assertion in the test case fails, while a test error is raised when the test case throws an unhandled exception or fails to complete. We also carefully checked the corresponding release notes, API documents, and migration guides (for Android) of the corresponding version pairs, and present the results in Column 8 of Table 5.2.

From Table 5.2, we make our observation as follows. Considering that cross-version testing may generate an under approximation of the number of BBIs, the prevalence of BBIs may be much higher than what is shown in the table.

Table 5.2: BBIs in Software-Library Version Pairs

Subject	Failure		Error		B-Incomp.		
	T.	I/A	T.	I/A	T.	I/A	Doc.
OpenJDK	203	2/2	15	2/2	35	2/2	13
Android	112	2/2	11	2/2	56	2/2	20
log4j	21	2/2	0	0/2	4	2/2	1
maven	14	3/4	226	4/4	19	4/4	3
bukkit	15	2/6	31	3/6	7	4/6	0
beanutils	0	0/1	0	0/1	0	0/1	0
codec	4	1/1	6	1/1	6	1/1	3
fileupload	0	0/3	12	2/3	2	2/3	0
commons-io	4	1/4	2	1/4	3	2/4	0
ela. Search	36	4/7	98	3/7	24	4/7	5
http-core	60	5/6	15	4/6	32	5/6	10
jodatime	15	5/7	6	2/7	17	5/7	7
jsoup	54	9/10	2	1/10	36	9/10	3
neo4j	3	2/5	7	1/5	6	2/5	0
snakeyaml	108	8/8	14	4/8	49	8/8	17
Tot.	**649**	**46/ 68**	**445**	**28/ 68**	**296**	**52/ 68**	**82**

Observation 1: BBIs between version pairs are prevalent among Java software libraries. We detect 296 BBIs in 14 of 15 subjects (93.3%), and 52 of 68 version pairs (76.5%). Averagely each version pair suffers from 4.4 BBIs, and only 82 of the 296 BBIs are documented

Distribution of BBIs between / within major versions. Beyond the overall status of backward incompatibilities between consecutive version pairs of software libraries, we further studied the difference between major and minor version pairs. The results are presented in Table 5.3. From Table 5.3, we have the observation as follows.

Observation 2: Major version pairs and minor version pairs suffered from 4.7 and 3.5 backward incompatibilities on average, respectively, and 76% of both types of version pairs are backward incompatible. Since BBIs are still prevalent within a major version, continuous minor version updates are still not safe, although they are slightly safer than major version upgrades.

Table 5.3: Distribution of BBIs in Different Version Pairs

Subject	Total		Average		Incomp. V / All V	
	Mj.	Mn.	Mj.	Mn.	Mj.	Mn.
JDK	23	12	23	12	1/1	1/1
Android	56	N/A	28	N/A	2/2	N/A
log4j	3	1	3	1	1/1	1/1
maven	10	9	5	4.5	2/2	2/2
bukkit	3	4	0.8	2	3/4	1/2
beanutils	N/A	0	N/A	0	N/A	0/1
codec	6	N/A	6	N/A	1/1	N/A
fileupload	0	2	0	1	0/1	2/2
commons-io	3	N/A	0.8	N/A	2/4	N/A
ela.Search	0	24	0	6	0/3	4/4
http-core	15	17	5	5.7	2/3	3/3
jodatime	17	N/A	2.4	N/A	5/7	N/A
jsoup	31	5	4.4	1.2	7/7	3/4
neo4j	6	0	2	0	2/3	0/2
snakeyaml	49	N/A	4.9	N/A	10/10	N/A
Total	**222**	**74**	**4.7**	**3.5**	**36/47**	**16/21**

5.3.3 Categorization of BBIs

To answer **RQ3**, we categorize BBIs according to their incompatible behaviors, invocation conditions, and the reasons why library developers brought them in.

For the categorization of incompatibilities from 3 aspects (behaviors, invocation constraints, and reasons), it is exploratory so we do not have a criterion available beforehand. We predefined high-level categories (e.g. return value change as a high-level category of incompatible behaviors). Then, first 2 authors went through the incompatibilities separately and classify them to the categories. They also annotate each BBI with labels made up by themselves. Then, all authors discussed and merged labels with similar meanings, and removed too-narrow labels to get the final set of finer-grained categories. If we found a finer-grained category cannot be put into predefined high-level categories (e.g., Environment in Figure 5.2), we made them separate high-level categories. Due to the complexity of BBIs, the categorization process is not easy, especially when the BBI involves multiple API methods.

Consider Example 1, which is a test code sample from Jsoup 1.7.1. In the test case, an

HTML document object is generated from HTML text, and then the document is printed out using `doc.body().html()` after setting char set to ascii and turn on escape mode. In Jsoup 1.7.3, the translation of some special characters for escape under ascii char set becomes different, so the printed HTML text will be changed. After inspection, we found that the change is made in method `html()`. Therefore, until `html()` is called, the memory status remains the same for both versions. In such a scenario, we determine that `html()` is the API method for the BBI, and the four API methods called before it are invocation constraint of the BBI.

Example 1 Identification of BBI-Related API Method

```
Document doc =
    Jsoup.parse("<p title=p> & < > ...");
doc.outputSettings().charset("ascii");
doc.outputSettings().escapeMode();
assertEquals("...", doc.body().html());
```

Incompatible Behaviors

From the 296 BBIs, we identified the following major categories of incompatible behaviors.

Exceptions and Crashes indicates that, in the new version of the software library, an API method throws exceptions in a different way. This category contains 4 sub-categories. *New Exception* indicates that the API method throws an exception in the new version but not in the old version. *Different Exception* indicates that the API method throws exceptions both versions, but the exceptions are different. *No Exception* indicates that, the API method throws an exception in the old version, but not in the new version. *Infinite Loop* indicates infinite loop in the new version.

Return Variable Change indicates that, the return value of an API method changes in the new version under certain usage scenario and input. Specifically, we divide this category into four sub-categories. *Value Change* indicates that a primitive value (e.g., integer, boolean, String) is changed. The BBI in Example 1 belongs to this category. *Field Change* indicates a field of the return object is changed. *Type Change* indicates that the actual type of the return value is changed, although the signature itself remain unchanged. This typically happens when the return type in the API method

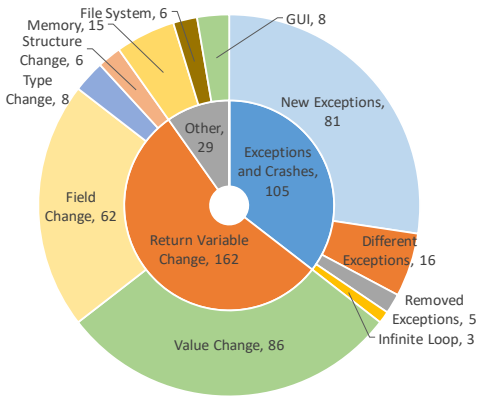

Figure 5.1: BBI Distribution on Incompatible Behaviors

signature has many subtypes (e.g., `java.lang.Object`). *Structure Change* indicates that, no primitive values in the return object is changed, but the object is organized differently (i.e., values of reference-type fields or sub-fields of the return object are changed).

Other Effects indicates that an API method causes a different side effect on other parts of the software itself or the operating system, such as value changes of other variables in *Memory*, changes of the *GUI*, and *File System*.

The distribution of 296 BBIs is presented in Figure 5.1. From the figure, we can see that the categories of *Return Variable Change* and *Exception and Crash* account for 162 and 105 BBIs, respectively, and they combined to account for more than 90% of the BBIs. The two major sub-categories for *Return Variable Change* are changes of primitive return value or field value changes of object-type return values, which account for more than 90% of the category. Such a distribution implies either most BBIs cause exceptions / simple return value changes, or such BBIs are more likely to be detected by regression testing. We will try to answer this question to some extent with our field bug study, but in either case, the following observation is true.

Observation 3: Most BBIs detected by regression testing will cause either exceptions or value changes of the return variable or its fields, while side effects and environment effects are seldom detected.

Invocation Constraints

We further investigated the conditions under which BBIs can be invoked, and identified the following five major types of such conditions.

Always indicates that the BBI always happens as long as the corresponding API method is invoked.

Error indicates that the BBI happens only when an error happens. An example is the change of error message when a network error is invoked.

Environment indicates that the BBI happens only under certain environments of the application (e.g., operating systems, language settings).

Multiple APIs indicates that a number of other API methods must be invoked before the backward incompatible API method to invoke the BBI. Example 1 belongs to this category.

Input indicates that the BBI happens when a certain input value is fed into the corresponding API method. Specifically, we divide this category into five sub-categories. *Trivial Value* indicates a null pointer or an empty string / list as the input. *String Format* indicates that strings with specific structure as the input. *Special Field* indicates that objects with specific values at a certain field as the input. *Special Value* indicates that certain primitive values (not including strings) as the input. **Special Type** indicates the argument of the API method must be of a specific subtype of its parameter type.

The distribution of 296 backward incompatibilities in the above categories is presented in Figure 5.2. We can observe that, the top 3 categories of invocation conditions are *Always*, *Specific Value*, and *String Format*. The commonality among the 3 categories of BBIs is that they can be invoked with one API method invocation. By contrast, the BBIs requiring multiple API methods to invoke are not common in those detected by regression testing.

Reasons for Bringing in BBIs

Since the library developers revised the test cases to accommodate the changes, we can find out the reasons and the purposes of the behavior change from the revised test code. We identified

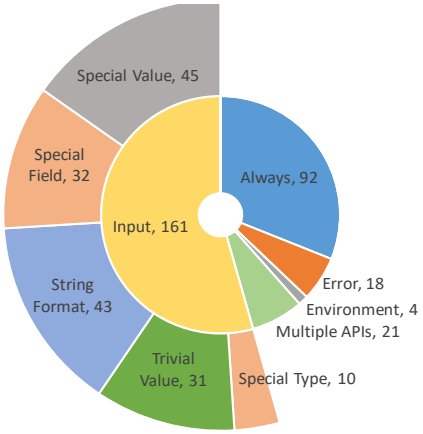

Figure 5.2: BBI Distribution on Invoking Conditions

the following 3 major categories of reasons, which contains 13 sub-categories.

Usage Change indicates that the library developers expect to keep the normal behavior of the library, but also expect client developers to change their usage pattern. This category has 4 sub-categories, which are *API Pattern Change* indicating changes on expected API sequences, *Enforce Rules* indicating rejecting of some poor input, *Enable Poor Inputs* indicating the support to some poor input, and *Input Format* indicating the change on the format of string type inputs.

Better Output indicates that the library developers expect to change the normal behavior of the API, while not expecting client developers to change their input. This category also has 5 sub-categories, which are *More Reasonable Output/Effect* indicating behavior change of an API method under a normal input to make it more reasonable[3], *Change Default Setting* indicating changing of the default setting (often a constant such as the default screen size), *Error Message* indicating change of reported errors, *Explicit Report* indicating explicitly throwing exceptions for errors, and *Output Format* indicating format change of string output.

Other indicates reasons not in the above 2 categories, including *Exposure of Internal Structure Change*, *Signature Change exposed with Reflection*, and *Upgrade Library*, in which the first two categories are regression faults.

The distribution of 296 backward incompatibilities in the above categories is presented in Fig-

[3]For example, in log4j 2.0.2, the time stamp on the log is changed from when the log is initialized to when the time stamp line is printed.

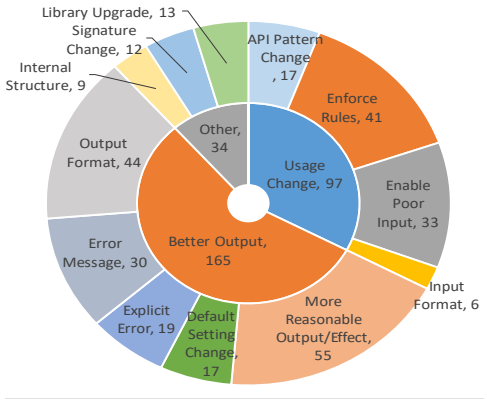

Figure 5.3: BBI Distribution on Reasons

ure 5.3. From the table, we can see the top 3 reasons for bringing incompatible behaviors are *More Reasonable Output/Effect*, *Output Format*, and *Enforce Rules*. We also find that, *Enable Poor Inputs*, the opposite of *Enforce Rules*, is the 4th popular reasons. Such contradiction reflects hesitation of library developers on whether responsibilities (e.g., input validation checking) should be at the library side or client side. Also, we did observe library developers moving back-and-forth on some incompatibilities. For example, in SnakeYaml, whether the dump output should include class name of the data value is changed 3 times through versions 1.3 to 1.6. Furthermore, although *More Reasonable Output/Effect* is the largest sub-category in *Better Output*, 93 of 165 BBIs in the category are not semantic changes, but presentation changes (e.g., error message, explicit exceptions, output formats). The 17 BBIs on change of default setting also related to client developers' preference. To sum up, we have the following observation.

Observation 4: Developers bring in a large portion of BBIs because they want to allow more or less inputs, or change the output presentation or option. This implies that the root cause of most BBIs is the different requirement of client developers, so the BBIs can be avoided if library developers understand requirement better (e.g., through surveys or code statistics on client projects), or have design for dual support.

5.4 Real-world Bug Study

In this section, to answer the **RQ4** and **RQ5**, we present the study result on real world bug reports related to BBIs, and explore how BBIs are affecting the client software developers. The basic information of the collected bug reports are presented in Table 5.4.

5.4.1 Collection of Bug Reports

To collect bug reports caused by BBIs. We searched two large on-line open bug repositories: JIRA and GitHub, on their project-specific bug report search engine. Specifically, we used as keywords the combination of terms related to software upgrading (e.g., "upgrade", "update", "version"), and the names of the software libraries listed in Table 5.1. From the collected bug reports, we *randomly* selected 500 bug reports, and carefully inspected these bug reports. During the inspection, we read the developers' comments and other references from the bug report to check whether they are confirmed by the developers to be caused by BBIs, and retain all the bug reports that are caused by BBIs.

We collected only bugs that are closed before Jan. 1st 2015 and never re-opened after that. We believe that these bugs are not likely to be re-opened so their status should be confirmed. We collect both bugs that are closed and fixed and bugs that are closed but not fixed due to developers' decision.

The reason is that, BBIs bugs are related to both software libraries and client software, so they can be fixed either at the library side or at the client side. Also, there are cases that a BBI bug is never fixed because the software library developers refuse to revert their changes, and the client developers did not find a way to work around it. In such cases, the developers may choose to not to support the new version of software library (not likely for runtime libraries such as OpenJDK and Android because not supporting the updated platform will cause dysfunction in client software), or have the users to tolerate the bug if the bug is relatively minor.

With the process above, we collected 126 bugs, and divided these bugs into two groups: *library bugs* that are submitted to software library projects that have BBIs, and *client bugs* that are

Table 5.4: Basic Information of Bugs

Subject	Library Bugs	Client Bugs	Total
Java SDK	8	10	18
Android	13	64	77
Other	29	2	31
Total	50	76	126

submitted to software client projects because they triggers BBIs of their software libraries. The breakdown of collected bugs is shown in Table 5.4.

From the table, we can observe that, as we used a random selection of bugs, the majority of selected bug reports are from Android and Java SE. The reasons are two fold. First, Java SE and Android are much popular than other software libraries studied. Second, Java SE and Android are both runtime platforms, so that client software developers do not have control on which version of JVM and Android system their software will be executed on. Therefore, BBIs may be revealed after a Java or Android update at the users' side, and get reported to the client software developers. This also explains another observation that, the majority of bugs of Android and Java SE are client bugs, while the majority of bugs of other libraries are library bugs, because Android and Java BBIs are more likely to be reported by end users, while BBIs in other libraries are more likely to be reported by client software developers to library software as library bugs. Sampling from unbalanced bug sets is difficult. Random sampling will be ruled by dominating classes (e.g., JVM and Android). Giving quota to classes, samples will not reflect the actual data distribution, and proper quota size needs to be determined. We chose random sampling in our study to see the actual impact of BBIs in the field.

5.4.2 Study on Bug-Inducing BBIs

In this subsection, we categorize the bug-inducing BBIs and compare them to the BBI distribution in our library study. Specifically, among the 126 bugs, we find 13 bugs (12 client bugs from Android and 1 library bug from http-core) that can be directly mapped to the BBIs found in our library study. This shows that the test cases in regression testing are far from sufficient for identifying all bug-inducing BBIs, but it also implies that, if the regression testing results can be

Table 5.5: Bug-Causing BBIs

Subject	Cause Library Bugs	Cause Client Bugs	Total
JDK	8	10	18
Android	13	51	64
Other	29	1	30
Total	50	62	112

well documented or conveyed to client developers in better ways, some bugs can be avoided.

Before we perform more in-depth investigation, we first manually scanned the bug reports to detect *cross-software duplicate bug reports*. Duplicate bug reports inside a project are typically labeled and we did not select them when collecting our bug report set. However, different client software may fail due to a same BBI of a same library, so these client bug reports are "duplicate" with each other. After the duplicate-bug-report detection, we identified 112 BBIs as presented in Table 5.5.

Incompatible Behaviors

Example 2 Bug-408: Be able to configure as a default SMS app in KitKat (from WhisperSystems/-TextSecure)

```
In Android 4.4, SMS apps are no longer able
to send SMS to the SMS provider (rejected
silently) in the Android system, unless they
are reset to receive a broadcast
SMS_DELIVER_ACTION.
```

The breakdown of bug-inducing BBIs per incompatible behaviors is presented in Table 5.6. Note that, in our bug study, we use the same categories as defined in Section 5.3.3. For incompatible behaviors, we found 2 BBIs that cannot be put into any defined categories, so we add a new category *Sys. Event*, which indicates changes in system events (Example 2).

In the table, Columns 2-3 present the number of library bugs (denoted as L) and client bugs (denoted as C) from Android. Columns 4-7 present similar data for Java and Other subjects. Column 8 presents the total of each line. Since Android and JDK dominates our bug report study

Table 5.6: Categorization of Incompatible Behaviors

Behavior		Android		JDK		Other		All
		L	C	L	C	L	C	
Exception and Crash	New Exception	2	21	5	5	13	1	47
	Infinite Loop	0	0	1	0	0	0	1
Ret. Var. Change	Value Change	1	0	0	1	2	0	4
	Field Change	1	2	1	2	7	0	13
	Type Change	1	2	0	0	1	0	4
	Structure Change	0	0	0	1	3	0	4
Other Effects	Memory	3	4	0	0	1	0	8
	GUI	5	19	0	1	1	0	26
	File Sys.	0	2	0	0	1	0	3
	Sys. Event	0	1	1	0	0	0	2

set, in the rest of the paper, when comparing bug study results and library study results, we provide both overall library study results, and library study results for JDK and Android combined. From Table 5.6, we have the following observations.

First, *New Exception* and *Infinite Loop* account for 48 of the 112 (43%) BBIs, which is higher than their proportion in the library study (28% overall, 30% for JDK+Android). These behaviors may be more likely to be reported as bugs due to their severity.

Second, *Value Change* accounts for 4 BBIs (3%), which is much smaller than its share in library study (29% overall, 18% for JDK+ Android). But *Field Change* accounts for 13 BBIs, which is much more than *Value Change* despite its lower share in library study.

Third, categories *Different Exception* and *No Exception* do not cause any bugs in our data set. This may be because client developers tend to avoid exceptions in their code so they are not affected. This also indicates that such changes are safer at the library side.

Example 3 Bug-46:Bold ZeroTopPaddingTextView displays cut off on 4.4 (from derekbrameyer/android-betterpickers)

```
In Android 4.4, a method that update the
padding setting must be invoked before
showing the date information, otherwise,
part of the date information can not be seen.
```

Fourth, GUI changes accounts for 26 BBIs (22.3%), which is the second largest category in bug study, but its share in library study is only 3% overall. The large number of GUI-change BBIs

Table 5.7: Categorization of Invocation Conditions

Conditions		Android		Java		Other		All
		L	C	L	C	L	C	
Always		5	14	2	1	2	0	24
Environment		0	1	0	0	1	0	2
Special Type		0	2	0	0	2	0	4
Multiple APIs		4	21	1	5	7	1	39
Input	Trivial Value	0	0	1	0	2	0	3
	String Format	2	2	3	3	6	0	16
	Specific Field	0	4	0	0	0	0	4
	Specific Value	2	7	1	1	9	0	20

may be related to the large proportion of Android-related bugs in our bug set. Although these BBIs may be specific to Android framework, they are still important and representative because of the popularity of Android apps, and the existence of similar frameworks. We discovered that, most bug-inducing GUI BBIs are changing settings of UI controls and thus affecting presentation. Examples of the settings include the position to put notification bars (top or bottom), box widths, etc. Example 3 presents a GUI-related bug. The bug report is caused by a BBI between Android 4.3 and Android 4.4 about the changed value of padding settings. It should be noted that, user interface bugs are not just decoration problems, and they may largely affect software usages (i.e., information cannot be seen, or buttons go outside the screen and cannot be clicked). In general, we have the observation as follows.

Observation 5: Distribution of BBIs in library study and field bug study is mismatched on the incompatibility behavior categories. New Exception, GUI Changes, and other side-effects account for a higher proportion in field bug study, while other categories of BBIs account for a lower proportion.

Invocation Constraints

The breakdown of bug reports according to BBI-invoking conditions is presented in Table 5.7. From the table, we have the following observations.

First of all, 24 of 112 (21%) BBIs always happen (compared to 31% overall and 22% in JDK+Android in the library study), causing at least 15 client-side bugs. Second, only 2 of the BBIs are related to the environment. This may be largely due to the platform independence of Java, so we doubt whether this conclusion can be generalized to other programming languages. Third, 39 BBIs (35%) occur only after certain other API methods are invoked and thus belong to the category of *Multiple API*, compared to 7% overall and 11% JDK+Android in the library study. This actually implies that a lot of BBIs happen under special usage scenarios which are not covered by the library-side test code. In such cases, client code may be a good source to extract suitable test cases for detecting BBIs in a software library. Fourth, no BBIs fall into the *Error* category of invocation-condition, which shows that BBIs in this category are less likely to cause client bugs or the client bugs they cause are difficult to detect. In general, we have the observation as follows.

Observation 6: Distribution of BBIs in library study and field bug study is mismatched on invocation constraints. Multiple APIs account for a much higher proportion in field bug study, compared to its share in library study. This implies that regression testing may need to be strengthened for usage patterns involving multiple API methods.

5.4.3 Documentation Study

To answer the third research question, we further studied the documentation status of the bug-inducing BBIs. Since these BBIs are bug inducing, we predict that they may be more poorly documented then the BBIs detected from cross-version testing (34% documented), and the results shown in Table 5.8 confirm our guess. In Table 5.8, the first column presents the documentation status (and the place if documented). The rest columns are organized similar to Table 5.6.

From Table 5.8, we can see that bug-inducing behavioral BBIs are very poorly documented. Only 14 (13%) BBIs are documented. Also, the documented changes are relatively scattered, especially for Android (in release notes, JavaDocs, and Migration guides). Also, 8 client bugs of Android and 4 client bugs of Java are related to documented behavioral changes. This implies that we may need a better way than documentation to convey the information of behavioral change and

Table 5.8: Documentation Status of BBIs

Behavior		Android		Java		Other		All
		L	C	L	C	L	C	
No Doc.		12	43	7	6	29	1	98
Doc.	Release Notes	1	1	1	3	0	0	6
	JavaDoc	0	3	0	1	0	0	4
	Migration Guide	0	4	0	0	0	0	4

Table 5.9: Resolution of Library Bugs

Resolution		Android	Java	Other	All
Fixed	Reverted	1	0	2	3
	Patched	6	4	16	26
	Double Support	0	0	1	1
Not Fixed	Intended	5	2	10	17
	Discouraged	1	2	0	3

remind client software developers about such changes.

5.4.4 Bug Resolution Study

To answer **RQ5**, we further studied how the real world bugs related to BBIs are resolved (note that they may be not fixed). Cross-software duplicate bugs may be fixed differently in different client software, so we view them as separate bugs in this subsection.

Resolution of Library Bugs

The breakdown of bugs according to how they are resolved is shown in Table 5.9. The first two columns present the types of resolution. If a library bug is fixed, we check whether it is fixed by a simple revert of the previous change, a patch of the previous change, or library developers decided to support both the previous behavior and the new behavior (typically by adding a parameter, and set either the previous behavior or the new behavior as default). If a library bug is not fixed, we study how library developers response to the bug report, and check whether it is intended behavior, or the developer is reporting a behavioral change on internal APIs which should not be used by client developers.

From Table 5.9, we have the following observations. First, 20 of 50 library bugs are not fixed. The major reason is that the behavior change described in the bug report is intentional. It should be noted that, since these behaviors are reported as library bugs, they may already cause some bugs or at least test failures at the client side, although the client bug may not be reported. Second, among the 30 bugs that are fixed, most of them are patched, which shows that the many bug-inducing BBIs are caused by side effect of other productive changes.

Resolution of Client Bugs

The breakdown of client bugs according to how they are resolved is shown in Table 5.10. The first two columns present the types of resolution. If a client bug is fixed, we check whether it is fixed by (1) changing the incompatible API method to another one; (2) changing the arguments of the incompatible API method to other constants, variable, or expressions; (3) adding an API invocation to set a certain internal-state field before or after the invocation of the incompatible API method; (4) converting the return value of the incompatible API invocation to the original value; (5) a global structural code change; (6) updating libraries; (7) changing configuration of software; or (8) bypassing the incompatibility behavior by skipping software features (see Example 4).

Example 4 Bug-969: Android 5.0 crash when trying to open the app (from open-keychain/open-keychain)

```
The cause of the bug is that, "ResourceNotFound
Exception" is thrown in Android 5.0 when an
"overall scroll glow drawer" is requested.
In the fix, the client developers simply catch
the exception without doing anything with it.
Therefore, the request of "overall scroll glow
drawer" is actually bypassed.
```

For the client bugs that are not fixed, we discovered two resolutions. The first resolution is that the client developer simply decided to wait until a new version library is released. One reason of such resolution is that, the BBI is caused by a regression bug, so the client developer waits for the library developers to release a bug-free version. Another reason (and the major reason in our

Table 5.10: Resolution of Client Bugs

Resolution		Android	Java	Other	All
Fixed	Change API	1	2	0	3
	Change Input	13	0	1	14
	Add Set Field	17	1	0	18
	Return Convert	6	0	0	6
	Structural	8	5	0	13
	Config	2	0	0	2
	Lib. Update	2	0	0	2
	Bypass	4	0	0	4
Not Fixed	Wait Lib. Fix	4	2	0	6
	Tolerate	7	0	1	8

study) is that, the BBI affects a third-party library that the client developers are relying on. Since the client developers cannot change the code of the third-party library (sometime they even do not have access to the source code), they are not able to resolve the BBI, and have to wait for the new version of the third party library. The second solution is that, the client developer simply tolerate the behavior change (if the BBI does not cause crashes). They may simply ask their users to get used to the new behavior such as a UI change, or transfer the BBI to downstream developers.

Table 5.9 shows that 14 client bugs are not fixed. Note that, we find that most developers are willing to and have tried to fix the bugs, but BBI-related bugs are more difficult to fix, because they typically involve code written by other people. Regarding the fixed client bugs, we have the observation as follows.

Observation 7: 41 of the 62 fixed client bugs are fixed through small changes including changing API, changing input value, add an API to set field, or convert the return value to the original value.

We present an example of client bug fixing with "Add an API to set field" in Example 5. The corresponding BBI is that, for Android 5.0, if the developer wants to start a service with an intent, the type of the intent must be explicitly set with the method `setClass(...)`, and the method `startService(...)` will check the `class` field of the intent and throw exceptions if the value is null.

Example 5 Fix: Bug-812: Lollipop notification settings won't work (from klassm/andFHEM)

```
 Intent intent = new Intent(
 Actions.NOTIFICATION_SET_FOR_DEVICE);
+intent.setClass(context,
     NotificationIntentService.class);
 intent.putExtra(BundleExtraKeys.DEVICE_NAME
     , deviceName);
 ...
 context.startService(intent);
```

Reporting to Library Developers.

In our study, we further studied whether client developers would like to report their bugs to the library developers. Among the 76 client bug reports, we find that the symptom is reported to library developers in only 6 bug reports. In most of the cases, the developers simply search through the Internet to find a workaround. Also, for the 6 reported bugs, only 3 are fixed by the library developers, while the other 3 are rejected because the corresponding BBIs are intended behaviors.

5.5 Discussion

In this section, we discuss the lessons learned, limitations and the threats to our study, and regression faults.

5.5.1 Lessons Learned

The final goal of our study is to find actionable goals for library / client developers to avoid or resolve bugs caused by behavioral backward incompatibilities.

Avoidance of BBI Bugs

Enforcing Old Tests on Release. In our study, we are surprised by the large number of BBIs found with simple cross-version testing. Note that all tests come with the project and developers are supposed to run them every time they build the code. Our study shows that, most tests detecting

BBIs are changed accordingly or deleted when BBIs were brought in, so they can never detect the BBIs. Therefore, we suggest to enforce testing with old tests when releasing a new version. Such a feature can be added to IDEs or version control systems. It is unnecessary that all old tests pass but developers should provide document or workaround for failed tests, especially on minor version releases.

Augmenting Regression Tests. Our results in Figure 5.1 shows that, although 105 BBIs cause different exception / crash status and 86 BBIs cause primitive return value changes, the remaining 105 of 296 BBIs (36%) cause memory state change (e.g., certain field of the returned object) or other side effects (e.g., GUI, file systems). Such BBIs can be revealed only with extra API calls or side-effect checking. Furthermore, Table 5.1 shows that such BBIs cause 60 of 112 (54%) studied BBI-related bugs. Some side effects, such as file and UI change can be difficult to detect using normal assertions. Augmented regression tests with more advanced memory revealing assertions will detect more bug-inducing BBIs, and automatic test augmentation techniques such as Ostra [130] may be helpful.

BBI Recommendation. Our study reveals mismatches between the distribution of BBIs in various categories and the corresponding distribution of BBI-related bugs, which shows that certain categories of BBIs are more likely to result in bugs. For example, GUI changes as incompatible behaviors and multiple APIs as invocation conditions has a much higher population in the studied bug-related BBIs, and none of studied bugs are caused by BBIs related to different exceptions or changed error messages. Also, APIs that are frequently used, once have BBIs, are more likely to result in multiple client-side bugs, which is also observed in our study. Furthermore, our study also observed contradictory reasons of behavioral changes (e.g., allowing lousy input and enforce input rules) and several reverted or double-supported BBIs, which shows that developers are not always clear about the consequences of involving BBIs. Therefore, based on the category and API-usage frequency information (together with other factors such as major or minor version released, development status, etc.), a recommendation system can be helpful for developers when they make decisions on involving a BBI in a release.

Test Change Tracking. Our study shows that, for 267 of 296 BBIs (90.2%), developers change their test code to accommodate the behavior change. Therefore, change of test code on public APIs can be a sign of BBIs, and tracking test code changes may provide more information about the happening and evolution of BBIs.

Detection and Resolution of BBI-Related Bugs

BBI Notification. Our study shows that the documentation status of behavioral incompatibilities is very poor. Even when a behavioral change is documented, there are still many relevant client bugs. We believe that, advanced techniques on documentation of behavioral incompatibilities is in a great need and will help reduce many bugs related to backward incompatibilities. The technique should be able to directly check the client code (e.g., finding code clones of a failed library-side test case) and raise warnings about potential relevant behavioral backward incompatibilities.

Advanced GUI Testing. We find that, GUI behavior change is one of the major cause of BBI-related client bugs. Many of such bugs cannot be easily detected by normal assertions, such as the bugs on text invisibility due to color and size change of UI controls (Example 3). Some BBI bugs can be detected only with human eyes. Automatic oracle checking is straightforward for unit regression testing, but hard for GUI regression testing. This calls for more advanced user-interface checking techniques to support automatic regression testing of GUI applications.

Test Code Reference. Since developers change their test code to accommodate BBIs, the test code change can be used as examples of how to work around BBIs. When using certain API methods, client developers may consider watching the library-side test-code changes, or searching for relevant test-code changes for workarounds when resolving the BBIs from client side. The resource of test code changes can also be used in documentation, BBI notifications, or automatic BBI resolution tools for the client side.

Automatic Fixing of Client Bugs. Our study shows that, 67% bugs (41 of 62) fixed from client are based on simple changes such as replacing the input arguments, converting the return values, and add an invocation to a certain field-setting API method before or after the BBI API

invocation. Consider Example 5, where an Intent object's field needs to be set before it is used. In many such BBIs, a validation is added in library code to check the field (e.g., checking class field of Intent for accessible classes) and an exception is thrown there. The existence of such patterns shows possibilities that many BBI-related bugs can be fixed automatically by adding new change rules to automatic bug fixing tools.

5.5.2 Limitations and Threats

Limitations. First, we use cross-version testing to detect BBIs in software libraries. This result in an under-estimation of the number of BBIs between version pairs. Also, since we require all test cases pass in their original versions, the detected BBIs are biased to the intended behavioral changes (since the relevant test cases are already fixed). However, the major goal of our study on regression testing is to show the prevalence of BBIs, and we believe the above mentioned limitations do not affect our conclusion. Second, when studying BBI-related bugs, we searched the bug repositories with keywords such as "update". Bugs related to BBIs do not have obvious keywords, such as "deadlock" for concurrency bugs. In particular, some BBI-related bugs may stay in the software for a long time, and the client developers may not realize the root cause of the bug even after it is fixed. Thus, our selected bugs may be biased to those bugs that are easily found to be relevant to BBIs.

Threats to Validity. The major threats to internal validity of our study is the potential errors and mistakes in the process of building software and performing regression testing, studying the bugs, and doing the statistics. To reduce this threat, we carefully wrote all the tools we used, and manually checked the results for correctness. The major threats to external validity is that, our conclusion may hold for only Java software libraries, and the libraries under study. Furthermore, our conclusion may hold for only the bugs studied. Since our bug dataset is dominated by Android and Java, some conclusions (e.g., about GUI) may be specific to these libraries. To reduce this threat, we chose the most popular Java software libraries, as well as randomly chose the bugs to be studied.

5.5.3 Detailed Classification of BBIs

Intention of Behavior Changes. In our paper, we do not differentiate regression bugs from other BBIs and treat them exactly the same. Our second study shows that, both regression bugs and intentional BBIs cause real-world client bugs. Also, it is very hard to tell whether a behavioral change is intentional because an intentional behavioral change can have unexpected side effects.

Contract-based Classification of Behaviors. In our paper, we manually categorized incompatible behaviors and invocation constraints in an intuitive way. In future, we plan to apply analysis tools to categorize incompatibilities in a more formal manner. Specifically, we plan to categorize incompatibilities to precondition violations where the new upgraded function requires more from its inputs than the old one(e.g., an non-null input is required) and postcondition violations where the return values of the new function do not subsume the old ones, or the new function throws different exceptions or has different side effects.

5.6 Related Work

Our work is related to studies on software library evolution, summarization of library changes, and migration for library evolution.

5.6.1 Studies on Software Libraries Evolution

Researchers have noticed that software libraries are evolving frequently for various reasons such as bug fixing [92], adding features [47], and better UI support [122] [121]. A number of studies have been conducted on the evolution of software libraries. Dig and Johnson [32] carried out an empirical study on how developers refactor API methods. Raemaekers et al. [106] proposed a measurement of software-library stability which considers API method difference and code difference, and studied the stability of 140 industrial Java systems based on the measurement. McDonnell et al. [85] studied the stability of Android APIs (in terms of added and removed classes and methods), and the time lag between the release of API changes and the corresponding adaptation at the client software side. Espinha et al. [39] interviewed 6 web client software de-

velopers and conducted an empirical study on four widely used web services to understand their API evolution trends, including the frequency of API changes, and the time given client developers to upgrade to the new version of services. Bavota et al. [16, 17], studied the evolution of software dependency upgrades in the apache software ecosystem. Robbes et al. [109] studied the reaction of developers to deprecated APIs in SmallTalk ecosystem. Wu et al. [128, 129] studied the evolution of API usages in large software ecosystem. Brito et al. [23]'s empirical study shows that deprecation message is not helpful for developers. The existing research efforts mainly focus on signature-level API changes (Raemaekers et al.'s work further considers the amount of code difference) to measure API changes and stability. By contrast, our study focuses on behavioral changes of software libraries, which are more difficult to be detected and may cause more severe consequences.

There have also been a number of research efforts on the impact of software-library to client software. Linares-VÃ¡squez et al. [75] studied the relationship between change proneness of APIs methods and the successfulness of client software that uses those API methods. Bavota et al. [18] further extends the work with more detailed experimental results. These research efforts shows that backward incompatibilities have much effect on the successfulness of client software, and thus motivate the study in our paper.

5.6.2 Summarizing Software Library Changes

To provide more information about the changes on API methods, there have also been research efforts trying to summarize changes between two consecutive versions of a software library. On the signature level, Wu et al. [126] proposed AUCA, an auditor for API changes, that reports a large variety of signature-level changes of APIs. Tang et al. [118] proposed to use tree adjoining language to summarize dependency relationships in libraries. Moreno et al. [89] proposed ARENA, an automatic tool to summarize software-library changes and generate release notes.

On the behavior level, McCamant and Ernst [83, 84] proposed to represent behavior API methods with program invariants generated with Daikon. Person et al. [101, 102] proposed differential

symbolic execution to summarize as symbolic expressions of inputs the semantic difference between two versions of a method. Lahiri et. al [68] proposed SymDiff, a tool that leverages a modularized approach to check semantic equivalence of different code versions, and calculate program paths that can reveal code behavioral difference.

5.6.3 Support for Library Migration

Godfrey and Zou [48] proposed a number of heuristics based on text similarity, call dependency, and other code metrics, to infer evolution rules of software libraries. Later on, S. Kim et al. [65] further improved their approach to achieve fully automation. M. Kim et al. [64] inspected existing framework evolution process to gather a number name-changing patterns and used these patterns to infer rules of framework evolution. Dagenais and Robillard [30] proposed *SemDiff*, which infers rules of framework evolution via analyzing and mining the code changes in the software library itself. Wu et al. developed *AURA* [127], which further involves multiple rounds of iteration applying call-dependency and text-similarity based heuristics on the code of software library itself. Most recently, Meng et al. [87] proposed *Hima*, which further enhances *AURA* by involving information from comments of code commits between two consecutive versions of software libraries.

5.7 Conclusion

In this paper, we present a study on behavioral backward incompatibilities based on regression testing of 68 version pairs of 15 Java software libraries, and inspection of 126 real world bugs. From our study, we find that behavioral backward incompatibilities are prevalent among Java software libraries, and caused most of real-world backward-incompatibility bugs. Furthermore, many of the behavioral backward incompatibilities are intentional, but are rarely well documented. We also categorized behavioral backward incompatibilities according to incompatible behaviors and invocation conditions, and found category mismatches between the the BBIs detected in regression testing and the BBIs causing bugs.

Chapter 6: LESSON LEARNED

From our large scale empirical study on the usage of mocking frameworks in software testing, we have learned that mocking frameworks are widely used in practice, and a small portion of dependencies are mocked - developer most likely to mock network, database and time consuming services API. The reason of mocking is that software dependencies such as web services and databases are very slow when they are invoked. Thus involving such dependencies in testing will slow down the whole testing process, which may be fine for system testing, but not acceptable in unit testing and regression testing which are typically performed whenever a change is committed.

Regression performance testing is an important but time/resource consuming process. Developers need to detect performance regressions as early as possible to reduce their negative impact and fixing cost. But in the evolution of a code it is very challenging that a code commit may include any type and scope of code changes, from one line revision, to feature addition and interface revision. A code commit may contain newly added code, especially new loops. No execution information of such code is available, but given that loops can have high impact on performance, there is a strong need of estimating the code commits execution time and frequency. Even if the execution time of changed code in a code commit has little impact on performance, the code commit may include changes on collection variables, eventually affecting the performance of unchanged code. From our study, we can conclude that it better to replace mocking with real code when code change performance impact is high.

Our finding from BBIs study shows that BBIs are prevalent among Java software libraries and majority of BBI bugs are not documented. There is little difference between major version pairs and minor version pairs suffered from backward incompatibilities on average, so they are equally important for BBI testing. Furthermore, majority client bugs are fixed through small changes including changing API, changing input value, add an API to set field, or convert the return value

to the original value which signifies that majority of BBI can be fixed automatically. Cross version testing is an important way to find BBI bugs bolster that developer should be careful when to mock and when not.

Chapter 7: FUTURE DIRECTIONS

7.1 Prioritization Regression Tests

The challenge of static analysis without profiling and input may not may not accurately assess the risk of sophisticated performance regression issues such as resource contention, caching effect. Improve the accuracy our cost model include context sensitive profiling information. Our current work consider single threaded program because of thread context switch, it is hard to capture execution time accurately. It would be better to consider thread contention as a feedback to our model to handle multi threaded program.

To provide more information about the changes on API methods, there have also been research efforts trying to summarize changes between two consecutive versions of a software library. On the signature level, Wu et al. [126] proposed AUCA, an auditor for API changes, that reports a large variety of signature-level changes of APIs. Moreno et al. [89] proposed ARENA, an automatic tool to summarize software-library changes and generate release notes. There is a good research direction to combine with our approach to make better hybrid model.

On the behavior level, McCamant and Ernst [83, 84] proposed to represent behavior API methods with program invariants generated with Daikon. Person et al. [101, 102] proposed differential symbolic execution to summarize as symbolic expressions of inputs the semantic difference between two versions of a method. Lahiri et. al [68] proposed SymDiff, a tool that leverages a modularized approach to check semantic equivalence of different code versions, and calculate program paths that can reveal code behavioral difference.

There is no research to prioritize regression tests based on backward incompatibility. Behavioral Backward incompatibility is our another research study where we categorize the behavioral bug beyond api signature change. It would be excellent research direction to prioritization regres-

sion tests based on backward incompatibility.

7.2 Augmenting Regression Tests

Software engineers use regression testing to validate software as it evolves. To do this cost-effectively, they often begin by running existing test cases. Existing test cases, however, may not be adequate to validate the code or system behaviors that are present in a new version of a system. Test suite augmentation techniques address this problem, by identifying where new test cases are needed and then creating them with reuse of existing test cases for augmentation. This is because existing test cases provide a rich source of data on potential inputs and code reachability, and existing test cases are naturally available as a starting point in the regression testing context. To create effective test suite augmentation techniques we need to understand the influence of the foregoing factors. Based on such an understanding, It would be better to create augmentation techniques that leverage test cases in a cost-effective manner. Input of test case are foreign element that effect the performance, how an existing test-generation tool to generate new test inputs to augment the existing test suite would be an excellent idea to go forward.

Our existing performance model can be extended easily by correlating input to collection or array. We can easily model the input change and amplify performance change that depends on input. It will be good tool to tune application performance on the average case and also for performance debugging. Behavioral bug like file change and UI layout change can be difficult to detect using normal assertions. Augmented regression tests with more advanced memory revealing code and assertions will detect more bug-inducing BBIs, and automatic test augmentation techniques such as Ostra [130] may be helpful. So augmented test suite would be better approach to expose more behavioral bug automatically.

7.3 Logging

Inappropriate provisioning of resources may lead to unexpected performance bottlenecks or memory overflow. So there is an essential need to system monitoring. Monitoring a computer on which System Monitor is running can affect computer performance slightly. Therefore, either log the System Monitor data to another disk (or computer) so that it reduces the effect on the computer being monitored, or run System Monitor from a remote computer. Using the logging trace System can predict resource usage and integrating with our performance model would be able to predict more sophisticated performance impact.

BIBLIOGRAPHY

[1] Apache commons maths. http://commons.apache.org/proper/commons-math/.

[2] Semantic versioning, http://semver.org/. Accessed: 2016-08-22.

[3] Xalan for java. https://xml.apache.org/xalan-j/.

[4] Yesterday, my program worked. today, it does not. why? In *International Symposium on the Foundations of Software Engineering*, pages 253–267. 1999.

[5] Testing at the speed and scale of google. `http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html`, 2011.

[6] Tools for continuous integration at google scale. `http://www.youtube.com/watch?v=b52aXZ2yi08`, 2011.

[7] Criticism of windows vista. `https://en.wikipedia.org/wiki/Criticism_of_Windows_Vista`, 2017.

[8] PerfRanker project web. `https://sites.google.com/site/perfranker2017/`, 2017.

[9] Sougou. `https://play.google.com/store/apps/details?id=com.sohu.inputmethod.sogou&hl=en`, 2017.

[10] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 74–89, 2003.

[11] Azzah Al-Maskari, Mark Sanderson, and Paul Clough. The relationship between ir effectiveness measures and user satisfaction. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 773–774, 2007.

[12] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering*, pages 432–441, 2005.

[13] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[14] Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance*, pages 292–301, 1993.

[15] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. Model-based performance testing (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 872–875, 2011.

[16] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 280–289, 2013.

[17] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, pages 1–43, 2014.

[18] G. Bavota, M. Linares-Vasquez, C. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. The impact of API change- and fault-proneness on the user ratings of android apps. *Software Engineering, IEEE Transactions on*, (99):1–1, 2014.

[19] Robert V Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[20] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering*, pages 106–115, 2004.

[21] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[22] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, pages 3–8, 2012.

[23] G. Brito, A. Hora, M. T. Valente, and R. Robbes. Do developers deprecate APIs with replacement messages? a large-scale analysis on java systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 360–369, 2016.

[24] Renée C. Bryce and Atif M. Memon. Test suite prioritization by interaction coverage. In *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*, DOSTA '07, pages 1–7, New York, NY, USA, 2007. ACM.

[25] Malcolm C. Post congress tristesse. In *TeX90 Conference Proceedings*, pages 84–89. TeX Users Group, March 1991.

[26] Bihuan Chen, Yang Liu, and Wei Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, pages 49–60, 2016.

[27] Yih-Farn Chen, D. S. Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. In *Proceedings of 16th International Conference on Software Engineering*, pages 211–220, 1994.

[28] P. K. Chittimalli and M. J. Harrold. Re-computing coverage information to assist regression testing. In *In International Conference on Software Maintenance*, pages 164–173, 2007.

[29] Roberta Coelho, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. Unit testing in multi-agent systems using mock agents and aspects. In *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, pages 83–90. ACM, 2006.

[30] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of International Conference on Software Engineering*, pages 481–490, 2008.

[31] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *Proceedings of the 4th International Workshop on Software and Performance*, pages 94–103, 2004.

[32] D. Dig and R. Johnson. How do APIs evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2):83–107, March 2006.

[33] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[34] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *ISSRE*, pages 113–124, 2004.

[35] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 235–245. ACM, 2014.

[36] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 102–112, 2000.

[37] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 102–112, New York, NY, USA, 2000. ACM.

[38] E. Engstr and P. Runeson. A qualitative survey of regression testing practices. In *In Product-Focused Software Process Improvement*, pages 3–16. Springer-Verlag, 2010.

[39] T. Espinha, A. Zaidman, and H.-G. Gross. Web API growing pains: Stories from client developers and their code. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 84–93, 2014.

[40] King Chun Foo. Automated discovery of performance regressions in enterprise applications. In *Master's Thesis*, 2011.

[41] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *2010 10th International Conference on Quality Software*, pages 32–41, 2010.

[42] Gregory Fox. Performance engineering as a part of the development life cycle for large-scale software systems. In *Proceedings of the 11th International Conference on Software Engineering*, ICSE '89, pages 85–94, 1989.

[43] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 178–188. IEEE, 2012.

[44] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. jmock: supporting responsibility-based design with mock objects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 4–5. ACM, 2004.

[45] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, not objects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246. ACM, 2004.

[46] Stefan J Galler, Andreas Maller, and Franz Wotawa. Automatically extracting mock object behavior from design by contract specification for test data generation. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 43–50. ACM, 2010.

[47] M. W. Godfrey and Q. Tu. Evolution in open source software: a case study. In *Proceedings 2000 International Conference on Software Maintenance*, pages 131–142, 2000.

[48] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, February 2005.

[49] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 156–166, 2012.

[50] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the 34th International Conference on Software Engineering*, pages 244–254. IEEE Press, 2012.

[51] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 127–139, 2009.

[52] Maurice H. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[53] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 145–155, 2012.

[54] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. Test input reduction for result inspection to facilitate fault localization. *Automated Software Engineering*, 17(1):5, 2009.

[55] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[56] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 60–71, 2014.

[57] Mainul Islam and Christoph Csallner. Dsc+ mock: A test case+ mock class generator in support of coding against interfaces. In *Proceedings of the Eighth International Workshop on Dynamic Analysis*, pages 26–31. ACM, 2010.

[58] Bo Jiang, Zhenyu Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 233–244, Washington, DC, USA, 2009. IEEE Computer Society.

[59] 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*, pages 77–88, 2012.

[60] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: Performance bug detection in the wild. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 155–170, 2011.

[61] T. Kalibera, L. Bulej, and P. Tuma. Automated detection of performance regressions: the mono experience. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 183–190, 2005.

[62] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 17–26, 2010.

[63] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering*, pages 119–129, 2002.

[64] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of International Conference on Software Engineering*, pages 333–343, 2007.

[65] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *Proceedings of Working Conference on Requirement Engineering*, pages 143–152, 2005.

[66] Bogdan Korel, Luay Ho Tahat, and Mark Harman. Test prioritization using system models. In *ICSM*, pages 559–568, 2005.

[67] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. Mantis: Automatic performance prediction for smartphone applications. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, pages 297–308, 2013.

[68] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. RebÃªlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification*, pages 712–717, 2012.

[69] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, 2003.

[70] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, 2003.

[71] A. W. Leung, E. Lalonde, J. Telleen, J. Davis, and C. Maltzahn. Using comprehensive analysis for performance debugging in distributed storage systems. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 281–286, 2007.

[72] H. K. N. Leung and L. White. Insights into regression testing. In *In International Conference on Software Maintenance*, pages 60–69, 1989.

[73] Ondřej Lhoták and Laurie Hendren. *Scaling Java Points-to Analysis Using Spark*, pages 153–169. 2003.

[74] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Trans. Software Eng.*, 33(4):225–237, 2007.

[75] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. API change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487, 2013.

[76] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 911–922, 2016.

[77] Tongping Liu and Xu Liu. Cheetah: Detecting false sharing efficiently and effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 1–11, 2016.

[78] Qi Luo. Automatic performance testing using input-sensitive profiling. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1139–1141, 2016.

[79] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 25–36, 2016.

[80] Alexey G. Malishevsky, Joseph R. Ruthruff, Gregg Rothermel, and Sebastian Elbaum. Cost-cognizant test case prioritization. Technical report, 2006.

[81] Madhuri R Marri, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on*, pages 149–153. IEEE, 2009.

[82] M. Mayer. In search of a better, faster, stronger web.

[83] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 287–296, 2003.

[84] S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *European Conference on Object-Oriented Programming*, pages 440–464, 2004.

[85] T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, pages 70–79, 2013.

[86] Lijun Mei, Zhenyu Zhang, W. K. Chan, and T. H. Tse. Test case prioritization for regression testing of service-oriented business applications. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 901–910, New York, NY, USA, 2009. ACM.

[87] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 353–363, 2012.

[88] M. MITCHELL. Gcc performance regression testing discussion.

[89] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora. Automatic generation of release notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 484–495, 2014.

[90] S. Mostafa and X. Wang. An empirical study on the usage of mocking frameworks in software testing. In *2014 14th International Conference on Quality Software*, pages 127–132, 2014.

[91] S. Mostafa and X. Wang. A statistics on usage of java libraries. In *Technical Report*, 2014.

[92] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85, 2002.

[93] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *In International Conference on Software Testing, Verification, and Validation*, pages 21–30, 2011.

[94] A. Nistor, P. C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 902–912, 2015.

[95] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 562–571, 2013.

[96] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 128–137, 2003.

[97] Rohan Padhye and Koushik Sen. Travioli: A dynamic analysis for detecting data-structure traversals. In *Proceedings of the International Conference on Software Engineering*, page To Appear, 2017.

[98] Heekwon Park, Seungjae Baek, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Regularities considered harmful: Forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 181–192, 2013.

[99] Benny Pasternak, Shmuel Tyszberowicz, and Amiram Yehudai. Genutest: a unit test and mock aspect generation tool. *International journal on software tools for technology transfer*, 11(4):273, 2009.

[100] Kochaar Pavneet Singh, Tegawendé François D Assise Bissyande, David Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In *Proceedings of the 13th International Conference on Quality Software (QSIC 2013)*. Nanjing, China, 2013.

[101] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 226–237, 2008.

[102] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 504–515, 2011.

[103] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. Creating a shared understanding of testing culture on a social coding site. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 112–121. IEEE, 2013.

[104] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing of concurrent classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 13–25, 2014.

[105] C. Pyo, S. Pae, and G. Lee. Dram as source of randomness. *Electronics Letters*, 45(1):26–27, 2009.

[106] S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 378–387, 2012.

[107] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 215–224, 2014.

[108] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[109] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to API deprecation?: The case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 56:1–56:11, 2012.

[110] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997.

[111] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *ICSM*, pages 179–188, 1999.

[112] David Saff and Michael D Ernst. Mock object creation for test factoring. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 49–51. ACM, 2004.

[113] P. Sawyer, P. Rayson, and R. Garside. REVERE: Support for requirements synthesis from documents. *Information Systems Frontiers*, 4(3):343–353, March 2002.

[114] Kai Shen, Ming Zhong, and Chuanpeng Li. I/o system performance debugging using model-driven anomaly characterization. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 23–23, 2005.

[115] Mark Sherriff and Laurie Williams. Empirical software change impact analysis using singular value decomposition. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 268–277, 2008.

[116] S. Stefanov. Yslow 2.0. In *CSDN Software Development 2.0 Conference*, 2008.

[117] Kunal Taneja, Yi Zhang, and Tao Xie. Moda: Automated test generation for database applications via mock objects. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 289–292. ACM, 2010.

[118] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–95, 2015.

[119] Richard J. Turver and Malcolm Munro. An early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice*, 6(1):35–52, 1994.

[120] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–, 1999.

[121] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-translate constant strings in web applications. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 87–96, 2010.

[122] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 16:1–16:11, 2012.

[123] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 457–466, 2010.

[124] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, 2000.

[125] Paul R. Wilson, Yves Bekkers, and Jacques Cohen. *Uniprocessor garbage collection techniques*, pages 1–42. 1992.

[126] W. Wu, B. Adams, Y.-G. Gueheneuc, and G. Antoniol. Acua: API change and usage auditor. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 89–94, 2014.

[127] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A hybrid approach to identify framework evolution. In *Proceedings of International Conference on Software Engineering*, pages 325–334, 2010.

[128] W. Wu, F. Khomh, B. Adams, Y. Guéhéneuc, and G. Antoniol. An exploratory study of API changes and usages based on apache and eclipse ecosystems. *Empirical Software Engineering*, 21(6):2366–2412, 2016.

[129] W. Wu, A. Serveaux, Y. Guéhéneuc, and G. Antoniol. The impact of imperfect change rules on framework API evolution identification: an empirical study. *Empirical Software Engineering*, 20(4):1126–1158, 2015.

[130] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, pages 380–403, July 2006.

[131] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 257–266. ACM, 2010.

[132] Dacong Yan, Guoqing Xu, and Atanas Rountev. Uncovering performance problems in java applications with reference propagation profiling. In *Proceedings of the 34th International Conference on Software Engineering*, pages 134–144, 2012.

[133] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[134] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 199–208, 2012.

[135] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: A change impact and regression fault analysis tool for evolving java programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 40:1–40:4, 2012.

[136] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 192–201, 2013.

[137] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 43–52, 2011.

[138] Hao Zhong, Lu Zhang, and Hong Mei. An experimental comparison of four test suite reduction techniques. In *Proceedings of the 28th International Conference on Software Engineering*, pages 636–640, 2006.

[139] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.