TOWARDS FASTER SOFTWARE REVISION TESTING

by

Lingchao Chen

Dedicated to my family.

TOWARDS FASTER SOFTWARE REVISION TESTING

by

LINGCHAO CHEN, BS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2021

ACKNOWLEDGMENTS

Foremost, I would like to express my deepest appreciation to my advisor, Professor Lingming Zhang. In the past five years, he continuously led me to the top research field and taught me how to present our ideas on paper. He is so brilliant and remarkable. Under his supervision, I completed a meaningful educational career. He deserves a round of applause.

I am also deeply indebted to my co-advisor Professor D.T Huynh. He extended a great amount of assistance and joined the entire process for my graduate research.

I would like to extend my sincere thanks to my committee members, Professor Cong Liu and Professor Wei Yang, for their suggestions and comments to help me improve the quality of my research work.

I am also grateful to Professor Sarfraz Khurshid, Professor Yuqun Zhang, Professor Xiaoyin Wang, Professor Foyzul Hassan, Professor Junjie Chen, Professor Jiajun Jiang, Mengshi Zhang, and Yiling Lou. They were the collaborators during my PhD study. They were so bright and hard-working and always gave me practical suggestions. I learned a lot from them. I hope I still have a chance to collaborate with them in the future.

I also want to thank Professor Xia Li and Samuel Benton from our Software Testing Group, for giving me such a delightful working environment during my PhD study. They supported me a lot in the past five years.

Finally, I would like to thank my family for their support, love, and trust. Without their support, I would not be able to complete my PhD study. Special thanks to my parents and girlfriend. This work is dedicated to them.

October 2021

TOWARDS FASTER SOFTWARE REVISION TESTING

Lingchao Chen, PhD
The University of Texas at Dallas, 2021

Supervising Professor: Dung Huynh, Chair

Software systems have been increasingly prevalent in all facets of our lives over the last few decades and play a critical role in modern living. They have a significant impact on the quality of our lives and provide tremendous convenience. However, software faults (also known as bugs) are unavoidable throughout the development of software systems that can have a substantial negative impact on the commercial company and result in significant losses. Numerous researchers have been working on this problem to test software systems during development and fix bugs after the software systems are established. However, due to the complexity of these systems, these approaches can be very time consuming. For example, mutation testing is an important component of software testing which can be very powerful to evaluate the quality of the test suite, but it can be extremely time consuming due to a large number of mutant execution. Also, Automated Program Repair (APR) techniques can reduce software debugging human efforts by advising plausible patches for buggy programs. However, the APR techniques need to repeatedly execute all the test suites to identify the plausible patches for the bugs under fixing. This process could be extremely costly. Therefore, it is essential to explore some approaches to speed up the processes of software testing and debugging.

In this dissertation, we aim to speed up software testing and debugging via faster software revision testing. The idea is to decrease the testing time between different revisions to speed up software testing and debugging. We explored two scenarios in software testing during the evolution of software systems: mutation testing and behavioral backward incompatibilities (BBIs) detection. We applied regression test selection (RTS) techniques to speed up mutation testing for the first study. Our study showed that both file-level static and dynamic RTS could achieve efficient and precise mutation testing, providing practical guidelines for developers. We called the second BBIs detection technique DeBBI which can reduce the end-to-end testing time for detecting the first and average unique BBIs by 99.1% and 70.8% for JDK compared to naive cross-project BBIs detection. Additionally, we detected 97 BBI bugs including 19 that were previously confirmed as unknown bugs. Lastly, we explored the application in patch validation of APR technique to speed up software debugging. We treated every single patch as a revision to develop a unified *on-the-fly* patch validation framework, named UniAPR. Our study demonstrated that *on-the-fly* patch validation could often speed up state-of-the-art source-code-level APR by over an order of magnitude, enabling all existing APR techniques to explore a more extensive search space to fix more bugs in the near future.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

With the rapid development of information technology, software systems have been broadly adopted in almost all aspects of modern lives. Software bugs are inevitable in modern software systems, affecting billions of people [45] and costing trillions of dollars in financial loss. In practice, software testing and debugging is essential to detect, localize, and fix bugs from software systems. However, software testing and debugging is quite expensive and time consuming, which utilizes 40%-50% of total resources, 30% of total effort, and 50%-60% of the total cost of software development [90, 125, 132, 143, 162]. Therefore, it is crucial to explore approaches to improve the efficiency of software testing and debugging. In this dissertation, we speed up software testing and debugging via faster software revision testing. We intend to improve the overall software testing and debugging efficiency by enhancing the speed between different revisions. In order to achieve this, we explored two scenarios in software testing and one scenario in software debugging. We worked on mutation testing and BBIs detection problems for software testing when faced with the evolution of software systems between different revisions. As for software debugging, we focused on speeding up the patch validation time of APR technique by treating each patch as a revision. We seek to speed them up by applying knowledge from various fields such as static program analysis, dynamic program analysis, and information retrieval.

Our dissertation consists of three sections. First, we suggested the idea of applying traditional RTS techniques for incrementally collecting mutation test results for evolving software systems to speed up mutation testing [52] (Section 1.1). Second, we proposed a technique named DeBBI [49] (Section 1.2) to detect BBI bugs between the different revisions when the system is upgraded. Third, we treated each patch generated by the APR technique as a revision. We aimed to speed up the validation time of each revision to reduce the total time utilized by the

APR technique. To achieve this, we introduced the first unified *on-the-fly* patch validation framework, UniAPR [50] (Section 1.3), to empirically study the impact of *on-the-fly* patch validation for state-of-the-art source-code-level APR techniques. We explored faster software revision testing in those three scenarios, implemented the three approaches in automated tools and evaluated them using extensively used benchmarks. The experimental results show the value of our techniques, and provide important guidelines to future developers and researchers.

1.1 Speeding up Mutation Testing via Regression Test Selection: An Extensive Study (ICST'18)

Mutation testing is one of the most powerful methodologies to evaluate the quality of test suites, and has also been demonstrated to be effective for various other testing and debugging problems, e.g., test generation, fault localization, and program repair. However, despite various mutation testing optimization techniques, mutation testing is still notoriously time-consuming. Regression Testing Selection (RTS) has been widely used to speed up regression testing. Given a new program revision, RTS techniques only select and rerun the tests that may be affected by code changes, since the other tests should have the same results as the prior revision. To date, various practical RTS tools have been developed and used in practice. Intuitively, such RTS tools may be directly used to speed up mutation testing of evolving software systems, since we can simply recollect the mutation testing results of the affected tests while directly obtaining the mutation testing results for the other tests from the prior revision. However, to our knowledge, there is no such study. Therefore, in Chapter 3, we perform the first extensive study (using 1513 revisions of 20 real-world GitHub Java projects, totalling 83.26 Million LoC) on the effectiveness and efficiency of various RTS techniques in speeding up mutation testing. Our study results demonstrate that both file-level static and dynamic RTS can achieve precise and efficient mutation testing, providing practical guidelines for developers.

1.2 Taming behavioral backward incompatibilities via cross-project testing and analysis (ICSE'20)

In modern software development, software libraries play a crucial role in reducing software development effort and improving software quality. However, at the same time, the asynchronous upgrades of software libraries and client software projects often result in incompatibilities between different versions of libraries and client projects. When libraries evolve, it is often very challenging for library developers to maintain the so-called backward compatibility and keep all their external behavior untouched, and behavioral backward incompatibilities (BBIs) may occur. In practice, the regression test suites of library projects often fail to detect all BBIs. Therefore, in Chapter 4, we propose DeBBI to detect BBIs via *cross-project* testing and analysis, i.e., using the test suites of various client projects to detect library BBIs. Since executing all the possible client projects can be extremely time consuming, DeBBI transforms the problem of cross-project BBI detection into a traditional information retrieval (IR) problem to execute the client projects with higher probability to detect BBIs earlier. Furthermore, DeBBI considers project diversity and test relevance information for even faster BBI detection. The experimental results show that DeBBI can reduce the end-to-end testing time for detecting the first and average unique BBIs by 99.1% and 70.8% for JDK compared to naive cross-project BBI detection. Also, DeBBI has been applied to other popular 3rd-party libraries. To date, DeBBI has detected 97 BBI bugs with 19 already confirmed as previously unknown bugs.

1.3 Fast and Precise On-the-fly Patch Validation for All (ICSE'21)

Generate-and-validate (G&V) automated program repair (APR) techniques have been extensively studied during the past decade. Meanwhile, such techniques can be extremely time-consuming due to the manipulation of program code to fabricate a large number of patches and also the repeated test executions on patches to identify potential fixes. PraPR, a recent G&V APR technique, reduces such costs by modifying program code directly at the level of compiled JVM bytecode with *on-the-fly patch validation*, which directly allows multiple bytecode patches to be tested within the same JVM process. However, PraPR is limited due to its unique bytecode-repair design, and is basically unsound/imprecise as it assumes that patch executions do not change global JVM state and affect later patch executions on the same JVM process. In Chapter 5, we propose a unified patch validation framework, named UniAPR, to perform the first empirical study of on-the-fly patch validation for state-of-the-art source-code-level APR techniques widely studied in the literature; furthermore, UniAPR addresses the imprecise patch validation issue by resetting the JVM global state via runtime bytecode transformation. We have implemented UniAPR as a publicly available fully automated Maven Plugin. Our study demonstrates for the first time that on-the-fly patch validation can often speed up state-of-the-art source-code-level APR by over an order of magnitude, enabling all existing APR techniques to explore a larger search space to fix more bugs in the near future. Furthermore, our study shows the first empirical evidence that vanilla on-the-fly patch validation can be imprecise/unsound, while UniAPR with JVM reset is able to mitigate such issues with negligible overhead.

CHAPTER 2
BACKGROUNDS AND RELATED WORKS

2.1 Mutation Testing

Mutation testing is a fault-based testing methodology for evaluating the quality of test suites, which was firstly proposed by DeMillo et al. [59] and Hamlet [76]. Given a program $\mathcal{P}$ under analysis, mutation testing applies a set of *mutation operators* to generate a set of *mutants* $\mathcal{M}$ for $\mathcal{P}$. Each mutation operator applies a transformation rule (e.g., negating a conditional statement from `if (x>0)` to `if (x<=0)`) to generate mutants; each mutant $m \in \mathcal{M}$ is the same as the original $\mathcal{P}$ except the mutated program statement. Then, all the mutants in $\mathcal{M}$ are executed against the test suite $\mathcal{T}$ of $\mathcal{P}$ to evaluate its effectiveness – for each mutant m, when the execution of $t \in \mathcal{T}$ on m has different result from the execution of t on $\mathcal{P}$, m is killed by t; otherwise, m survives. In this way, mutation testing results can be represented as a *mutation matrix* [188]:

Definition 2.1.1 (Mutation Matrix)**.** *A mutation matrix is a function $\mathcal{MEM} : \mathcal{M} \times \mathcal{T} \rightarrow \{$**?**, ❍, ✓, ✗$\}$ that maps a mutant $m \in \mathcal{M}$ and a test $t \in \mathcal{T}$ to: (1)* **?** *if t has not been executed on m and thus the result is unknown, (2)* ❍ *if the execution of t cannot execute the mutated statement in m (in this case m cannot be killed by test t and does not even need to be executed against the test), (3)* ✗ *if t executes the mutated statement but does not kill m, and (4)* ✓ *if t kills m.*

To illustrate, Figure 2.1 shows an example program and its corresponding test suite. The program is changed from an old revision into a new revision by modifying Class `C`. Following Definition 2.1.1, the mutation matrix for the old revision of the program is presented in Table 2.1. Given such mutation testing results, developers can get feedbacks about the limitations of the existing test suites from the surviving mutants. Based on the mutation

```
public class A {
  public int m1(int a){
    return a + 1;
  }
}
public class B extends A{
  public int m1(int a){
    if (a>0){
        return a + 2;
    }
    else{
        return C.m3(a);
    }
  }
  public int m2(int a){
    return a − 1;
  }
}
public class C{
  public static int m3(int a){
- ~~return a+2;~~
+ return a+3;
  }
}
```

```
public class T1 {
  public void test1() {
    A a = new A();
    assertEquals(11,a.m1(10));
  }
}
public class T2 {
  public void test2() {
    A a = new B();
    assertTrue(a.m1(0)>=2);
  }
}
public class T3 {
  public void test3() {
    A a = new B();
    assertEquals(9,a.m2(10));
  }
}
public class T4 {
  public void test4() {
    A a = new B();
    assertEquals(12,a.m1(10));
  }
}
```

Figure 2.1: Example code

Table 2.1: Old version mutation test result

Mutant	Statement	Mutated Statement	T1	T2	T3	T4
m1	n3	`return a-1;`	✓	❍	❍	❍
m2	n3	`return 11;`	✗	❍	❍	❍
m3	n8	`if(a<0)`	❍	✗	❍	✗
m4	n16	`return a-2;`	❍	❍	✓	❍
m5	n21	`return (--a)+2;`	❍	✓	❍	❍

matrix, the ratio of killed mutants (i.e., *mutation score* [35]) can be easily computed and has been widely recognized as one of the most powerful methodologies for test suite evaluation. Note that in practice, *partial* mutation matrices [188], which aborts executing remaining tests against a mutant once the mutant is killed, have been widely used for computing mutation scores for sake of efficiency, since they return the same mutation scores as the original *full* mutation matrices. Besides its original application for evaluating test suite effectiveness, now it has also been successfully applied in various other testing and debugging problems, e.g., it has been applied for simulating real faults for testing experiments [38, 61, 87], guiding automated test generation [67, 135, 189], boosting fault localization [102, 123, 134, 190], and transforming code for automated program repair [58, 140, 149, 172].

2.2 Regression Test Selection

Regression test selection (RTS) [70, 77, 93, 131, 145, 151, 153, 186] is one of the most widely used approaches to speeding up regression testing. The main purpose of RTS is to reduce regression testing efforts by just re-running the tests affected by code changes, since the tests not affected by code changes should not change their outcomes in the new revision. In the literature, various dynamic and static RTS techniques have been proposed. Dynamic RTS techniques [70, 77, 131, 145, 151, 186] collect the test dependency information dynamically when executing tests on the old revision; then any tests whose dependencies overlap with code changes get selected. Although widely studied and used, dynamic RTS may not be suitable when dynamic test dependencies are not available, challenging to collect (e.g., code instrumentation for collecting dynamic test dependencies may cause timeouts or interrupt normal test run for real-time systems), or even unsafe (e.g., dynamic test dependencies for code with non-determinism may not cover all the possible traces, thus making RTS unsafe). Therefore, researchers have also proposed static RTS techniques [93, 98, 146, 153] to use static analysis to over-approximate the test dependencies. For both static and dynamic RTS, test dependencies and code changes can be computed at different granularities (e.g., file and method levels). Since prior work has demonstrated that file-level RTS performs the best for both static and dynamic RTS [70, 98], we introduce the basics of state-of-the-art file-level RTS techniques in the rest of this section.

2.2.1 Static RTS

STARTS [98] is state-of-the-art static file-level RTS technique based on *class firewall* analysis. STARTS computes the set of classes that might be affected by the changes and builds a "firewall" around those classes; then, any test classes within the class firewall can potentially be affected by code changes and are selected as affected tests. The notion of *firewall* analysis was first proposed by Leung et al. [99] to compute the code modules that might be affected by code changes. Then, Kung et al. [93] further proposed the notion of *class firewall* analysis by considering the characteristics of object-oriented languages, such as inheritance. STARTS performs class firewall analysis based on the Intertype Relation Graph (IRG) proposed by Orso et al. [131] to consider the specific features of the Java programming language. Formally, IRG can be defined as follows:

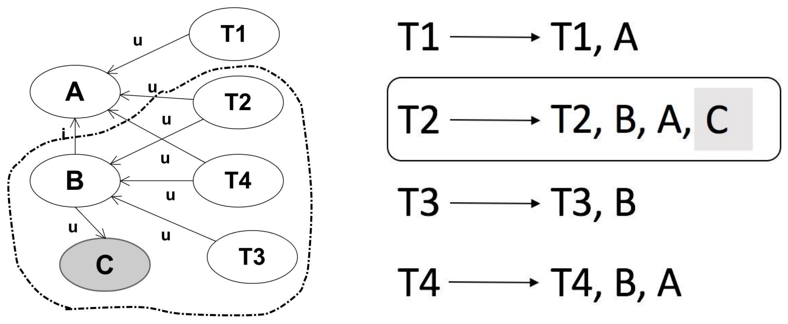
Definition 2.2.1 (Intertype Relation Graph). *The intertype relation graph of a given program is a triple $\langle \mathcal{N}, \mathcal{E}_i, \mathcal{E}_u \rangle$ where:*

- *$\mathcal{N}$ is the set of nodes representing all classes or interfaces;*
- *$\mathcal{E}_i \subseteq \mathcal{N} \times \mathcal{N}$ is the set of inheritance edges; there exists an edge $\langle n_1, n_2 \rangle \in \mathcal{E}_i$ if type n_1 inherits from n_2, or implements the n_2 interface;*
- *$\mathcal{E}_u \subseteq \mathcal{N} \times \mathcal{N}$ is the set of use edges; there exists an edge $\langle n_1, n_2 \rangle \in \mathcal{E}_u$ if type n_1 uses any element of n_2 (e.g., via field accesses and method invocations).*

Based on IRG, the class firewall can be computed as:

Definition 2.2.2 (Class Firewall). *The class firewall for a set of changed types $\tau \subseteq \mathcal{N}$ is computed over the IRG $\langle \mathcal{N}, \mathcal{E}_i, \mathcal{E}_u \rangle$ using as the transitive closure: $firewall(\tau) = \tau \circ \overline{\mathcal{E}}^*$, where $\circ$ is the relational product, $\overline{\mathcal{E}}$ denotes the inverse of all use and inheritance edges, i.e., $(\mathcal{E}_i \cup \mathcal{E}_u)^{-1}$, and * denotes the reflexive and transitive closure.*

To illustrate, class `C` is changed during software evolution for the example program in Figure 2.1. Here, we apply STARTS to select the affected tests to test the revision. Figure 2.2(a) shows the class firewall analysis results using STARTS. In the IRG, the use and inheritance edges are marked with labels "*u*" and "*i*", respectively. The dashed area is the class firewall analysis results, i.e., all the classes within the dashed area can potentially reach the changed class



(a) STARTS (b) Ekstazi

Figure 2.2: Static and dynamic RTS example

(marked in gray), and thus be affected by the changed class. In this way, Test `T2`, `T3` and `T4` are all within the class firewall, and are selected.

2.2.2 Dynamic RTS

Ekstazi [70] is state-of-the-art dynamic RTS technique based on file-level test dependencies. When executing the tests on the old program revision, Ekstazi performs on-the-fly bytecode instrumentation to record the class files used by each test. Then, given the new program revision, Ekstazi computes the changed class files via Checksum differencing, and selects any tests whose file-level dependencies involve the changed classes. Although a number of dynamic RTS techniques based on finer-granularity analysis have been proposed, they may incur large overhead due to the finer-grained analysis. The Ekstazi work [7] shows that compared with the method-level dynamic FaultTracer technique [186], Ekstazi can greatly save the end-to-end testing time, i.e., including both test execution time and RTS overhead. The Ekstazi tool now has been integrated with various build systems (including Ant, Maven, and Gradle), and has been adopted by practitioners from the Apache Software Foundation.

To illustrate, Figure 2.2(b) presents the RTS process using Ekstazi. Shown in the figure, for each test, Ekstazi dynamically traces the set of class files used by it in the old revision. Then, Ekstazi simply selects the tests that execute changed classes as the affected tests. In this example, only `T2` executes the changed class (marked in gray) in the old revision, and thus is selected for rerun. Note that Ekstazi may select less tests than STARTS since STARTS use static dependency information to over-approximate the potential test dependencies.

2.3 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 [152], 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. [103] proposed a 2-optimal strategy based on two different strategies: hill-climbing, and genetic programming. respectively. Jiang et al. [83] proposed an adaptive random strategy for test-case prioritization. Bryce and Memon [47] proposed 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. [184] proposed a generic strategy that has flavor of both total and additional strategies.

Besides proposing generic prioritization strategies, researchers have also investigated test prioritization using different levels of code coverage. There have been research work based on statement and branch coverage [152], function coverage [64], block coverage [62], modified condition/decision coverage [62], etc. There have also been research [91] on test-case prioritization using coverage of system models. Mei et al. [119] investigated criteria based on dataflow coverage for testing service-oriented software. More recently, Saha et al. [155] utilized the textual similarity between tests and code changes based on IR to perform test prioritization. In this dissertation, we are prioritizing client software projects instead of test cases, and thus we face two very different challenges. First, since it takes huge amount of time to execute tests of all client software projects, our approach must be static (i.e., not using any runtime information). Second, compared with test cases which are designed to cover different parts of a software project, client software projects contain much more redundancy. Therefore, to overcome these challenges, we developed an IR-based approach and further optimized it considering the diversity of term coverage (based on MMR) and test relevance (via extending static RTS).

2.4 Automated Test Generation

Another area that is related to our work is test generation based on existing client code. Suresh et al. [168] proposed an approach to automatically generate test cases by mining source code from client software projects, and later extended the technique with mining of dynamic execution traces [167, 169]. Bozkurt and Harman [46] proposed an approach to generate test cases from web service transactions. Pradel and Gross [139] combined specification mining from client code and test generation to detect API usage bugs. More recently, Ma et. al. [113] proposed to use library test cases to guide test-case generation for client software. Reiss [144] proposed to use semantic code search to find potential client code for test generation. Research efforts in this area focuses on generating test cases for one software project based on source code or test code of the current or other software projects. Therefore, they actually solve a different problem, and suffer from the general problems of test generation, such as the test oracle and unrealistic test (i.e., exploration of method invocation sequences that never happens in reality) problems, when directly used for library code testing. In contrast, our prioritization technique opens a new dimension via utilizing the large number of existing client project tests in the wild for detecting library BBIs, and can be complementary to these existing test generation techniques.

2.5 Automatic Program Repair

Automatic program repair (APR) aims to suggest likely patches for buggy programs to reduce the manual effort during debugging. The widely studied generate-and-validate (G&V) techniques attempt to fix bugs by first generating a pool of patches and then validating the patches via certain rules and/or checks [69, 85, 95, 107, 108, 115, 126, 175]. Generated patches that can pass all the tests/checks are called *plausible* patches. However, not all plausible patches are the patches that the developers want. Therefore, these plausible patches are further manually checked by the developers to find the final *correct* patches (i.e., the patches semantically equivalent to developer patches). G&V APR techniques [69, 84, 85, 95, 109, 115, 126, 175] have been extensively studied in recent years, since it can substantially reduce developer efforts in bug fixing. According to a recent work [105], researchers have designed various APR techniques based on heuristics [85, 95, 106], constraint solving [63, 117, 126, 180], and pre-defined templates [69, 88, 104]. Besides *automated* bug fixing, researchers have also proposed Unified Debugging [43, 110] to leverage various off-the-shelf APR techniques to help with *manual* bug fixing. In this way, the application scope of APR techniques has been extended to all possible bugs, not only the bugs that can be automatically fixed.

Meanwhile, despite the spectacular progress in designing and applying new APR techniques, very few techniques have attempted to reduce the time cost for APR, especially the patch validation time which dominates repair process. For example, JAID [51] uses patch schema to fabricate meta-programs that bundle several patches in a single source file, while SketchFix [81] uses sketches [100] to achieve a similar effect. Although they can potentially help with patch generation and compilation, they still require validating each patch in a separte JVM, and have been shown to be rather costly during patch validation [69]. More recently, PraPR [69] uses direct bytecode-level mutation and HotSwap to generate and validate patches on-the-fly, thereby bypassing expensive operations such as AST manipulation/compilation on the patch generation side as well as process creation and JVM warm-up on the patch validation side. This makes PraPR substantially faster than state-of-the-art APR (including JAID and SketchFix). However, PraPR is limited to only the bugs that can be fixed via bytecode manipulation, and can also return imprecise patch validation results due to potential JVM pollution.

2.6 Java Agent and HotSwap

A Java Agent [55] is a compiled Java program (in the form of a JAR file) that runs alongside of the JVM in order to intercept applications running on the JVM and modify their bytecode. Java Agent utilizes the instrumentation API [55] provided by Java Development Kit (JDK) to modify existing bytecode that is loaded in the JVM. In general, developers can both (1) *statically* load a Java Agent using `-javaagent` parameter at JVM startup, and (2) *dynamically* load a Java Agent into an existing running JVM using the Java Attach API. For example, to load it statically, the manifest of the JAR file containing Java Agent must contain a field `Premain-Class` to specify the name of the class defining `premain` method. Such a class is usually referred to as an Agent class. The Agent class is loaded before any class in the application class is loaded and the `premain` method is called before the main method of the application class is invoked. The `premain` method usually has the following signature: `public static void premain(String agentArgs, Instrumentation inst)`. The second parameter is an object of type `Instrumentation` created by the JVM that allows the Java Agent to analyze or modify the classes loaded by the JVM (or those that are already loaded) before executing them. Specifically, the `redefineClasses` method of `Instrumentation`, given a *class definition* (which is essentially a class name paired with its "new" bytecode content), even enables dynamically updating the definition of the specified class, i.e., directly replacing certain bytecode file(s) with the new one(s) during JVM runtime. This is typically denoted as the

JVM HotSwap mechanism. It is worth mentioning that almost all modern implementations of JVM (especially, so-called HotSpot JVMs) have these features implemented in them.

By obtaining `Instrumentation` object, we have a powerful tool using which we can implement a HotSwap Agent. As the name suggests, HotSwap Agent is a Java Agent and is intended to be executed alongside the patch validation process to dynamically reload patched bytecode file(s) for each patch. In order to test a generated patch during APR, we can pass the patched bytecode file(s) of the patch to the agent, which *swaps* it with the original bytecode file(s) of the corresponding class(es). Then, we can continue to run tests which result in executing the patched class(es), i.e., validating the corresponding patch. Note that subsequent requests to HotSwap Agent for later patch executions on the same JVM are always preceded by replacing previously patched class(es) with its original version. In this way, we can validate all patches (no matter generated by source-code/bytecode APR) on-the-fly sharing the same JVM for much faster patch validation.

CHAPTER 3
SPEEDING UP MUTATION TESTING VIA REGRESSION TEST SELECTION: AN EXTENSIVE STUDY

Mutation Testing [59, 76] was originally proposed to evaluate the quality of software test suites. During mutation testing, a number of *mutants*, each of which has small syntactic changes compared to the original program to simulate potential faults, will be generated based on a set of *mutation operators*, e.g., Removing Method Invocations (RMI) [10]. Then, each mutant will be executed against the test suite to check whether the test suite has different outcomes on the mutant and the original program. If the test suite can distinguish the two, the mutant is denoted as *killed*; otherwise, the mutant *survived.* Based on the correlation between mutants and real faults, if a test suite can kill more mutants, it may also detect more real faults.

Mutation testing is often considered as one of the most powerful methodologies in evaluating test suite quality [37, 66, 157, 185]. To date, mutation testing has been used for test-suite evaluation in a large number of software testing studies in the literature [82]; mutation testing is also gaining more and more adoptions among practitioners, e.g., mutation testing has been used by developers of The Ladders, Sky, Amazon, State Farm, Norways e-voting system, and the Linux kernel [10, 36, 54, 74]. Furthermore, mutation testing has also been successfully applied for various other testing and debugging problems, e.g., real fault simulation for software-testing experimentation [38, 61, 87, 111], automated test generation [67, 135, 189], fault localization [102, 123, 134, 190], and program repair [58, 140, 149, 172].

Despite the effectiveness of mutation testing, the application of mutation testing on real-world systems can still be quite challenging. One of the major challenges is that mutation testing can be extremely time consuming due to mutant execution – it requires to execute each mutant against the test suites under analysis. Researchers have proposed various techniques to speed up mutation testing, e.g., *selective mutation testing* that executes a subset of mutants to represent all the mutants [129, 161, 177, 183, 185], *weakened mutation testing* that executes each mutant partially [80, 178], and *predictive mutation testing* that predicts mutation testing results based on some easy-to-obtain features (without actual mutant execution) [182]. Although such techniques can speed up mutation testing to some extent, they may provide rather imprecise mutation testing results.

Researchers have also proposed the *regression mutation testing* (ReMT) technique [188] to speed up mutation testing of evolving software systems while providing precise mutation testing results (i.e., producing mutation results close to the traditional un-optimized mutation testing). The basic idea of ReMT is that the high cost of mutation testing can be amortized during software evolution. Based on dynamic coverage collection and static control-flow graph reachability analysis, ReMT incrementally updates the mutation testing results for a new revision based on the mutation testing results of an old revision. ReMT has been shown to be effective in reducing mutation testing cost, and can be combined with other optimization techniques for even faster mutation testing. However, due to the fine-grained analysis and complicated design, there is no practical tool support for ReMT.

Regression Test Selection (RTS) aims to speed up regression testing via only selecting and rerunning the tests that are affected by code changes during software evolution [70, 77, 93, 131, 145, 151, 153, 186]. A RTS technique is *safe* if it selects all the tests that may be affected by the code changes. A typical RTS technique computes test dependencies at certain granularity (e.g., method or file level) and then selects all the tests whose dependencies overlap with code changes. Various dynamic and static RTS techniques at different granularities have been proposed: while dynamic RTS techniques [70, 186] trace test dependencies dynamically via code instrumentation, static RTS techniques [98] use static analysis to over-approximate test dependencies. To date, various mature RTS tools (e.g., Ekstazi [7], STARTS [11], and

FaultTracer [187]) have been publicly available, and have been adopted by practitioners (e.g., Ekstazi has been applied to Apache Camel [4], Apache Commons Math [5], and Apache CXF [6]). Our key insight is that we can simply rerun the mutation testing results for the affected tests computed by such mature RTS tools for practical regression mutation testing. The reason is that the unaffected tests do not cover code changes, and thus may have the same mutation testing results as the prior revision. Therefore, in this paper, we perform the first extensive study on speeding up mutation testing via RTS techniques using 1513 revisions of 20 real-world GitHub Java projects, totalling 83.26 Million LoC. Since different RTS techniques may perform differently for mutation testing, we consider state-of-the-art dynamic and static, as well as file-level and method-level RTS techniques.

This paper makes the following contributions:

- **Insight** We propose the first attempt to directly apply RTS techniques for practical regression mutation testing.

- **Study**: We perform an extensive study of various practical RTS techniques (e.g., Ekstazi, STARTS, and FaultTracer) on speeding up mutation testing (using the PIT tool) of 20 open-source GitHub Java programs with 1513 revisions, totalling 83.26 Million LoC. To our knowledge, this is the largest scale study on mutation testing.

- **Findings**: We find that surprisingly both file-level static and dynamic RTS techniques can be used for precise and efficient regression mutation testing, while the dynamic method-level RTS tends to be less precise, providing important guidelines for practical mutation testing.

Algorithm 1: RTS-based mutation testing

```
Input: Old program $\mathcal{P}_1$, new program $\mathcal{P}_2$, test suite $\mathcal{T}$, old mutation matrix $\mathcal{MEM}_1$, partial matrix tag $Partial$
Output: New mutation matrix $\mathcal{MEM}_2$
1  begin
       // Initialize the new mutant execution matrix
2      $\mathcal{MEM}_2 \leftarrow \emptyset$
       // Perform RTS to selected affected tests
3      $\mathcal{T}_s \leftarrow RTS(\mathcal{P}_1, \mathcal{P}_2, \mathcal{T})$
       // Generate mutants for $\mathcal{P}_2$
4      $\mathcal{M}_2 \leftarrow MutGen(\mathcal{P}_2)$
5      for $m \in \mathcal{M}_2$ do
6          for $t \in \mathcal{T}$ do
7              if $t \in \mathcal{T}_s$ then
                   /* Execute the test against the mutant, and record results */
8                  $\mathcal{MEM}_2 \leftarrow Execute(t, m)$
9              else
                   /* Copy mutant execution results from the old version */
10                 $\mathcal{MEM}_2 \leftarrow \mathcal{MEM}_1(t, m)$
               /* Only enabled for collecting partial mutant execution matrix: abort test execution
                  against a mutant once the mutant is killed */
11             if $Partial \wedge \mathcal{M} \times \mathcal{T} = ✓$ then
12                 Break;
13     return $\mathcal{MEM}_2$ // Return the final mutation matrix
```

3.1 Approach

3.1.1 Overview

Algorithm 1 presents the overview of RTS-based regression mutation testing. The inputs include two program revisions during software evolution (i.e., $\mathcal{P}_1$ and $\mathcal{P}_2$), the regression test suite (i.e., $\mathcal{T}$), the mutation matrix collected on the old revision (denoted as $\mathcal{MEM}_1$), and the configuration argument indicating whether to collect partial mutation matrices (denoted as $Partial$). The output is the mutation matrix for the new program revision. Shown in the algorithm, Line 2 first initializes $\mathcal{MEM}_2$ as empty. Line 3 then performs RTS to select the affected tests. Note that the algorithm is general for Ekstazi, STARTS, or any other RTS techniques. Line 4 generates all the mutants for $\mathcal{P}_2$. Then, for each mutant of $\mathcal{P}_2$, the algorithm tries to collect its mutation testing results. For each test, the algorithm checks whether it is selected as affected tests. For each selected affected test, the algorithm runs the test on the mutant and store the mutant execution results in $\mathcal{MEM}_2$ (Lines 6-7). Note that most modern mutation testing tools (e.g., PIT [10], Major [9], and Javalanche [8]) will be

Table 3.1: New version mutation test result

Mutant	Statement	Mutant Statement	T1	T2	T3	T4
m1	n3	`return a-1;`	✓	❍	❍	❍
m2	n3	`return 11;`	✗	❍	❍	❍
m3	n8	`if(a<0)`	❍	✗	❍	✓
m4	n16	`return a-2;`	❍	❍	✓	❍
m5	n22	`return (--a)+3;`	❍	✗	❍	❍

able to skip executing a test on a mutant when the test does not cover the mutated statement of the mutant, and thus here we apply the same optimization. For each unselected test, the algorithm simply copies its mutant execution results from prior revision into $\mathcal{MEM}_2$ (Line 9). Note that Lines 10-12 are only enabled when collecting *partial* regression mutation matrix, i.e., aborting test execution for each mutant as soon as the mutant is killed. Finally, the algorithm returns the collected $\mathcal{MEM}_2$ as the resulting mutation matrix.

Note that although the algorithm is surprisingly simple, it actually supports both full and partial mutation testing. Furthermore, it handles various corner cases in practice: (1) when a test is newly added, a practical RTS tool (such as STARTS and Ekstazi) will always select it, thus its mutation testing results can be collected by the algorithm; (2) when a test is deleted during software evolution, it will not be within the current test suite $\mathcal{T}$, thus its mutation testing results will not be collected; (3) when a mutant is newly added due to code modifications, tests covering the code modifications will be selected, and thus its mutation testing results will be collected; (4) when a mutant is deleted due to code modifications, all the tests covering it will be selected, making it no longer available in the resulting mutation matrix. However, it does not mean that the algorithm always provides the same results as simply rerunning all tests on all mutants. We will provide detailed analysis in our next subsection.

3.1.2 Analysis and Illustration

We now illustrate the RTS-based regression mutation testing using the example program in Figure 2.1. Note that we only illustrate the full mutation matrix collection, since the partial

Table 3.2: Regression mutation testing via STARTS

Mutant	Statement	Mutant Statement	T1	T2	T3	T4
m1	n3	`return a-1;`	✓	❍	❍	❍
m2	n3	`return 11;`	✗	❍	❍	❍
m3	n8	`if(a<0)`	❍	✗	❍	✓
m4	n16	`return a-2;`	❍	❍	✓	❍
m5	n22	`return (--a)+3;`	❍	✗	❍	❍

Table 3.3: Regression mutation testing via Ekstazi

Mutant	Statement	Mutant Statement	T1	T2	T3	T4
m1	n3	`return a-1;`	✓	❍	❍	❍
m2	n3	`return 11;`	✗	❍	❍	❍
m3	n8	`if(a<0)`	❍	✗	❍	✗
m4	n16	`return a-2;`	❍	❍	✓	❍
m5	n22	`return (--a)+3;`	❍	✗	❍	❍

matrices can be collect in similar ways. The two program revisions and the test suite are already shown in Figure 2.1, the full mutation matrix for the old version has already been shown in Table 2.1. The expected full mutation matrix for the new revision (via running all mutants against all tests for the new revision) is shown in Table 3.1. Note that due to software revision, mutation testing results for mutant $m3$ have changed, while both mutant $m5$ itself and its mutation results have changed. We now show how STARTS and Ekstazi can be applied for regression mutation testing.

Shown in Section 2.2.1, the STARTS technique selects tests `T2`, `T3`, and `T4`. Therefore, for each mutant of the new revision, its execution results on `T1` can be directly copied from the old revision, while its execution results on all the other tests have to be recollected. In this way, we have the resulting mutation matrix shown in Table 3.2. Shown in the table, with STARTS, only 4 out of the 6 mutant-test cells need to be collected via mutation testing, and the mutation results for both mutants $m2$ and $m5$ (which are the only two mutants with different results in the two revisions) are correctly updated. On the contrary, shown in Section 2.2.2, the Ekstazi technique only selects test `T2` for the new revision. Therefore, for each mutant of the new revision, only `T2` needs to be executed for mutation testing. The resulting matrix is shown in Table 3.3. Shown in the table, with Ekstazi, only 2 out of the

6 mutant-test cells need to be recollected, even faster than regression mutation testing via STARTS. However, although Ekstazi is able to correctly update the mutation results for mutant $m5$, it fails to correctly update the mutation results for mutant $m3$, which should be killed by `T4`.

We further analyze the different performance by Ekstazi and STARTS in terms of both time savings and mutation testing precision. Shown in prior work [98], STARTS uses static analysis to over-approximate test dependencies, and selects any test that may potentially reach code changes. In terms of time savings, STARTS may perform worse than Ekstazi due to the more conservative selection (for both regression testing and mutation testing). This is also confirmed in our example, on which STARTS runs 4 mutant-test cells, while Ekstazi runs only 2. However, in terms of mutation testing precision, the conservative selection by STARTS may actually be beneficial. For two program revisions $\mathcal{P}_1$ and $\mathcal{P}_2$, if test t's dynamic dependencies (e.g., computed by Ekstazi) on $\mathcal{P}_1$ do not touch code changes between $\mathcal{P}_1$ and $\mathcal{P}_2$, t's execution on $\mathcal{P}_2$ should be the same as $\mathcal{P}_1$ (unless the code is non-deterministic), thus Ekstazi is safe for regression testing. However, t's execution on a mutant of $\mathcal{P}_1$ may be diverged (e.g., the mutant may negate a branch statement) and execute code changes. Therefore, the mutation testing results of t may change for $\mathcal{P}_2$, making Ekstazi unsafe for mutation testing. On the contrary, when test t's static dependencies (e.g., computed by STARTS) do not touch code changes, the execution of t on a mutant also may not touch code changes. The reason is that traditional mutation operators (e.g., all the mutation operators used by modern mutation tools such as PIT [10], Major [9], and Javalanche [8]) change program statements within method bodies, and usually cannot diverge program execution to the statically unreachable code. In our study, we further investigate the mutation testing precision issues in real-world systems.

Table 3.4: Projects used in study

ID	Project Name	SHA	Tests	Revs	kLOC
p1	HikariCP	980d8d	1922.31	92	14.89
p2	commons-io	fedbfc	99.00	74	55.43
p3	OpenTripPlanner	3cb177	138.55	86	145.81
p4	commons-functor	5d6b05	156.51	40	38.98
p5	commons-lang	013621	142.06	100	136.86
p6	commons-net	2b0f33	42.78	100	58.46
p7	commons-text	aa2a77	42.20	96	26.13
p8	commons-validator	4f60e5	69.04	97	31.12
p9	compile-testing	e4269a	9.57	73	6.07
p10	invokebinder	004d2f	3.07	100	4.31
p11	logstash-logback-encoder	4336fd	40.76	95	15.28
p12	commons-codec	1a4d9c	51.65	79	34.15
p13	commons-dbutils	633749	26.28	66	12.44
p14	commons-scxml	eac3f6	353.22	37	27.47
p15	commons-csv	ed6adc	14.18	89	10.15
p16	commons-jexl	f4babe	43.00	39	36.02
p17	la4j	db2041	39.00	39	59.64
p18	commons-cli	b486fb	24.48	96	11.28
p19	commons-math	79c471	436.48	72	316.22
p20	asterisk-java	684be6	39.00	40	81.98

3.2 Experiment Setup

In this section, we first described our dataset (Section 3.2.1), followed by our experiment settings (Section 3.2.2), and evaluation metrics (Section 3.2.3).

3.2.1 Dataset

We used 20 real-world Java projects from GitHub as our subject systems. The selected subject systems have been widely used on regression testing or mutation testing research [54, 72, 159, 182]. Following prior work on regression test selection [98], for each project, we start from the `HEAD` revision, and obtain the 100 most recent revisions. Then, we keep all the revisions on which we can successfully run (1) `mvn test`, (2) the used PIT mutation testing tool, and (3) the studied RTS tools (e.g., Ekstazi, STARTS, and FaultTracer). Table 3.4 shows the detailed information about the used projects. In the table, Columns 1 and 2 present the project IDs and names. Column "SHA" presents the SHA for the `HEAD` revision of each project, while Column "REVS" presents the number of revisions used for each project. Column "TESTS" presents the average number of tests for all the used revisions of each project. Finally, Column "kLOC" presents the average size information for all the revisions of each project. Shown in the table, the sizes of our subject systems range from 4.31K lines of the code (LoC) per revision (`invokebinder`, with 100 revisions) to 316.22K LoC per revision (`commonsmath`, with 72 revisions). In total, all the 1513 studied revisions have in total 83.26 Million LoC, and represent the largest experimental study for mutation testing to our knowledge.

3.2.2 Experiment Setting

In this study, we use the PIT [10] mutation testing tool (with all its 16 mutation operators in Version 1.1.5), since it has been demonstrated to be one of the most robust and efficient mutation testing tools for Java, and has been widely used in prior mutation-testing-related studies [54, 72, 159, 182]. The original PIT implementation aborts test execution for a mutant once it is killed and thus only supports partial mutation testing; to also support full mutation testing, following August et al. [159], we force PIT to execute each mutant against the remaining tests even after the mutant is killed. To study the effectiveness and efficiency of static/dynamic RTS tools on regression mutation testing, we apply STARTS and Ekstazi, since they have been shown to represent state-of-the-art RTS tools [70, 98]. Since both STARTS and Ekstazi are file-level RTS techniques, to further explore the impact of RTS with different test dependency granularities on regression mutation testing, we also study state-of-the-art dynamic method-level RTS technique, FaultTracer [186]. Note that we do not study the static method-level RTS since prior work has demonstrated that it is both imprecise and unsafe compared with the file-level RTS STARTS [98].

We now briefly describe our experimental setup. Our goal is to study the effectiveness and efficiency of various state-of-the-art RTS techniques on mutation testing. For each studied RTS technique on each subject system, we first obtain the SHA list of studied revisions of the subject system from GitHub. Then for every revision of the subject, we go through the following steps. (1) We make sure `mvn test` can pass all the tests. (2) We then perform full mutation testing using PIT (e.g., with command `mvn org.pitest-maven:mutationCoverage`) to get the actual mutation testing results for the revision. (3) We run the studied RTS techniques to select the affected tests for the revision (e.g., with command `mvn ekstazi:ekstazi` for Ekstazi, and command `mvn starts:starts` for STARTS). (4) We collect the regression mutation testing matrices via only rerunning the affected tests for each mutant while copying the results for other tests from the prior revision (Algorithm 1). (5) Finally, we compare our RTS-based mutation testing matrix with the actual mutation testing matrix to get effectiveness and efficiency metrics. For time reduction, we record both the RTS-based mutation testing time and the RTS overhead.

3.2.3 Evaluation Metrics

We now talk about the metrics to evaluate the effectiveness and efficiency of the RTS-based regression mutation testing.

Effectiveness

- **Error Cells** In the mutation matrix, each test has an execution result on each mutant. Thus, a test and a mutant can compose a mutant-test cell. The error cell metric measures the number of cells with different execution results between the actual and the RTS-based mutation matrices. To illustrate, there is one error cell ($m3$-`T4`) between Ekstazi mutation matrix (Table 3.3) and the actual new mutation matrix (Table 3.1). On the contrary, the regression mutation matrix collected by STARTS (Table 3.2) is exactly the same as the actual mutation matrix, and does not have any error cell.

- **Error Mutation Score** The users often use the mutation score information (e.g., the ratio of mutants killed by tests) to evaluate test effectiveness. Therefore, we also compute error mutation score as the difference in mutation scores of the actual mutation matrix and the mutation matrix collected via RTS, i.e., $|\mathcal{MS}_{actual} - \mathcal{MS}_{RTS}|$. For our example, 3 out of the 5 mutants are killed in the actual mutation matrix of the new revision, i.e., mutation score is 60%. When using both Ekstazi and STARTS, the regression mutation score is also 60%, indicating 0% error mutation score.

Efficiency

- **Test-level Reduction** shows the ratio of tests reduced by RTS. For example, Ekstazi and STARTS select 1 and 3 out of the 4 tests, respectively; thus, the test-level reduction for Ekstazi/STARTS is 75%/25%.

- **Mutant-level Reduction** indicates the ratio of mutants requiring at least one test execution. For example, for STARTS, 3 of the 5 mutants require at least one test execution, indicating a mutant-level reduction of 40%; for Ekstazi, 2 of the 5 mutants require at least one test execution, i.e., 60% reduction.

- **Cell-level Reduction** measures the ratio of mutant-test cells requiring execution. Note that we only consider the mutant-test cells where the tests executes the corresponding mutated statements, since the other cells cannot be killed and do not need execution. For example, for STARTS, 4 out of 6 mutant-test cells require execution, indicating a cell-level reduction of 33.3%; for Ekstazi, 2 out of 6 mutant-test cells require execution, indicating a cell-level reduction of 66.7%.

- **Time Reduction** All the above efficiency metrics do not consider the actual mutation testing time savings. Therefore, this metric measures the actual mutation testing time reduction.

Table 3.5: Effectiveness of Ekstazi and STARTS

Project Name	Mutation Score	Ekstazi						STARTS					
		Error Cells			Error Mutation Score			Error Cells			Error Mutation Score		
		min	max	avg	min	max	avg	min	max	avg	min	max	avg
commons-cli	79.05%	0	3621	128.08	0.0%	10.48%	0.4005%	0	2128	52.94	0.0%	6.26%	0.1941%
commons-validator	41.8%	0	3522	209.5	0.0%	11.32%	0.8371%	0	3286	198.92	0.0%	10.14%	0.7821%
commons-dbutils	50.36%	0	1026	107.22	0.0%	7.35%	0.9823%	0	1026	93.11	0.0%	7.35%	0.8215%
asterisk-java	12.95%	0	1109	324.03	0.0%	10.02%	2.2991%	0	993	299.85	0.0%	9.33%	2.2177%
commons-cli	84.43%	0	2355	85.42	0.0000%	0.00%	0.0000%	0	2355	83.37	0.0000%	0.00%	0.0000%
commons-dbutils	46.23%	0	3398	418.51	0.0000%	0.50%	0.0276%	0	3398	391.57	0.0000%	0.50%	0.0276%
commons-validator	67.73%	0	6191	97.35	0.0000%	0.00%	0.0000%	0	6191	97.39	0.0000%	0.00%	0.0000%
asterisk-java	17.88%	0	2	0.28	0.0000%	0.00%	0.0000%	0	2	0.28	0.0000%	0.00%	0.0000%

Table 3.6: Efficiency of Ekstazi and STARTS

Project Name	Results			Ekstazi			STARTS		
	Tests	Mutants	Cells	Test Level	Mutant Level	Cell Level	Test Level	Mutant Level	Cell Level
commons-cli	24.48	1926	5886	92.03%	91.14%	91.46%	90.09%	89.83%	89.35%
commons-validator	69.04	5573	7419	96.92%	97.05%	97.23%	96.46%	96.0%	96.18%
commons-dbutils	26.33	1146	1689	90.54%	90.27%	89.97%	86.52%	88.55%	86.4%
asterisk-java	39.0	12487	12721	96.78%	98.9%	98.88%	89.41%	98.78%	98.66%
commons-cli	24.48	2767	1480348	92.03%	90.36%	97.78%	90.09%	89.03%	97.25%
commons-dbutils	26.28	2037	1309638	90.68%	91.36%	99.02%	86.72%	89.41%	98.63%
commons-validator	69.04	6680	2858829	96.92%	94.97%	98.79%	96.46%	94.42%	98.79%
asterisk-java	57.00	18774	716054	96.78%	97.91%	96.18%	89.41%	96.54%	93.93%

3.3 Result Analysis

This section, we are working on the following four research questions.

3.3.1 RQ1: how do state-of-the-art static and dynamic RTS techniques perform in terms of regression mutation testing effectiveness?

Table 3.7 shows the effectiveness of Ekstazi and STARTS in regression mutation testing. In the table, Columns 1 and 2 present the project name and the average mutation score for all the revisions of each project. The mutation scores range from 6.77% to 84.43%, indicating that we cover a variety of test suites with different effectiveness for evaluating RTS-based mutation testing. Columns 3-5 and Columns 6-8 present the error cell and error mutation score metrics when using Ekstazi. Similarly, Columns 9-11 and Columns 12-14 present the error cell and error mutation score metric values when using STARTS. Based on the table, we have the following observations. First, surprisingly, the average error mutation score values across all projects are only 0.0423% and 0.0364% when using Ekstazi and STARTS,

Table 3.7: Effectiveness of Ekstazi and STARTS

Project Name	Mutation Score	Ekstazi						STARTS					
		Error Cells			Error Mutation Score			Error Cells			Error Mutation Score		
		min	max	avg	min	max	avg	min	max	avg	min	max	avg
HikariCP	23.24%	0	861	20.3	0.0000%	1.61%	0.0418%	0	197	2.16	0.0000%	0.51%	0.0056%
commons-io	32.03%	0	3479	289.32	0.0000%	0.07%	0.0052%	0	3921	449.67	0.0000%	0.09%	0.0131%
OpenTripPlanner	6.89%	0	107	8.26	0.0000%	0.14%	0.0109%	0	26	3.01	0.0000%	0.05%	0.0037%
commons-functor	6.77%	0	1	0.36	0.0000%	0.01%	0.0024%	0	1	0.36	0.0000%	0.01%	0.0024%
commons-lang	78.75%	0	6036	320.88	0.0000%	0.07%	0.0238%	0	6036	260.51	0.0000%	0.07%	0.0206%
commons-net	23.16%	0	3319	116.74	0.0000%	0.03%	0.0048%	0	3319	116.74	0.0000%	0.03%	0.0048%
commons-text	74.08%	0	322	19.47	0.0000%	0.08%	0.0106%	0	322	19.47	0.0000%	0.08%	0.0106%
commons-validator	67.73%	0	6191	97.35	0.0000%	0.00%	0.0000%	0	6191	97.39	0.0000%	0.00%	0.0000%
compile-testing	56.22%	0	704	19.19	0.0000%	0.14%	0.0038%	0	704	19.19	0.0000%	0.14%	0.0038%
invokebinder	42.65%	0	677	11.12	0.0000%	0.05%	0.0005%	0	677	16.64	0.0000%	0.05%	0.0005%
logstash-logback-encoder	54.24%	0	109	12.55	0.0000%	0.62%	0.1160%	0	192	18.21	0.0000%	0.38%	0.1108%
commons-codec	77.90%	0	262	19.71	0.0000%	0.08%	0.0170%	0	262	19.71	0.0000%	0.08%	0.0170%
commons-dbutils	46.23%	0	3398	418.51	0.0000%	0.50%	0.0276%	0	3398	391.57	0.0000%	0.50%	0.0276%
commons-scxml	44.50%	0	10275	314.22	0.0000%	0.47%	0.0558%	0	10275	301.08	0.0000%	0.47%	0.0325%
commons-csv	71.87%	0	2507	94.41	0.0000%	0.75%	0.1456%	0	2507	94.2	0.0000%	0.75%	0.1405%
commons-jexl	46.10%	0	8364	855.47	0.0000%	0.63%	0.0960%	0	8364	693.63	0.0000%	0.63%	0.0759%
la4j	57.62%	0	367	40.79	0.0000%	0.02%	0.0030%	0	17	1.54	0.0000%	0.01%	0.0007%
commons-cli	84.43%	0	2355	85.42	0.0000%	0.00%	0.0000%	0	2355	83.37	0.0000%	0.00%	0.0000%
commons-math	71.47%	265	60361	4894.31	0.0523%	0.57%	0.2800%	3	60361	4535.89	0.0026%	0.57%	0.2572%
asterisk-java	17.88%	0	2	0.28	0.0000%	0.00%	0.0000%	0	2	0.28	0.0000%	0.00%	0.0000%
Average	49.19%	13	5485	381.93	0.0026%	0.29%	0.0423%	0	5456	356.23	0.0001%	0.22%	0.0364%

Table 3.8: Effectiveness of mutation testing without RTS

Project Name	Mutation Score	Error Cells			Error Mutation Score		
		min	max	avg	min	max	avg
HikariCP	23.24%	5	5794	731.98	0.0000%	11.42%	0.6342%
commons-io	32.03%	9	8102	2897.43	0.0030%	0.74%	0.2899%
OpenTripPlanner	6.89%	0	107	12.61	0.0000%	0.13%	0.0143%
commons-functor	6.77%	0	43	1.46	0.0000%	0.26%	0.0214%
commons-lang	78.75%	14	13160	1137.78	0.0010%	0.41%	0.0256%
commons-net	23.16%	0	6466	248.81	0.0000%	0.40%	0.0132%
commons-text	74.08%	0	2395	265.85	0.0000%	1.58%	0.1226%
commons-validator	67.73%	0	12381	439.29	0.0000%	0.53%	0.0271%
compile-testing	56.22%	0	5971	413.96	0.0000%	16.77%	0.4462%
invokebinder	42.65%	0	1352	126.87	0.0000%	16.77%	0.5993%
logstash-logback-encoder	54.24%	3	743	122.62	0.0000%	2.33%	0.2397%
commons-codec	77.90%	0	1497	107.67	0.0000%	1.07%	0.0640%
commons-dbutils	46.23%	0	5184	890.42	0.0000%	3.06%	0.1805%
commons-scxml	44.50%	0	38543	22354.63	0.0000%	1.79%	0.4094%
commons-csv	71.87%	0	5002	306.98	0.0000%	1.88%	0.1766%
commons-jexl	46.10%	1755	143337	74685.79	0.0209%	1.96%	0.6944%
la4j	57.62%	8	43602	5846.15	0.0000%	2.02%	0.2297%
commons-cli	84.43%	0	4710	365.23	0.0000%	0.52%	0.0192%
commons-math	71.47%	750	102228	7398.85	0.0016%	2.86%	0.1520%
asterisk-java	17.88%	0	170	16.77	0.0000%	0.06%	0.0083%
Average	49.19%	127	20039	5919	0.0013%	3.33%	0.2183%

respectively. Similarly, the average error cell values are only 381.93 and 356.23 when using Ekstazi and STARTS, respectively. The observation demonstrates the effectiveness of using both Ekstazi and STARTS for regression mutation testing. The reason is that even using dynamic file-level RTS (such as Ekstazi), mutations that diverge test execution may not incur the test to touch a new class/file, making file-level RTS relatively safe for mutation testing. Second, we observe that STARTS has less error mutation score or error cell values than Ekstazi. As also shown in Section 3.1.2, the reason is that STARTS relies on static test dependencies computed via over-approximation which may not be diverged to touch code changes after mutation if the original static dependencies cannot reach code changes. Finally, even STARTS can also incur imprecision in mutation score (although it is negligible), while STARTS may not have any imprecision issue according to our analysis in Section 3.1.2. We look into the code, and find that the imprecision was due to the unsafe STARTS test selection incurred by the use of Java reflections (also observed in prior RTS study [98]). In summary, both dynamic and static file-level RTS can be used for precise regression mutation testing, and STARTS tends to be slightly more precise.

To further investigate the effectiveness of RTS-based mutation testing, we also study the effectiveness of directly copying entire mutation matrices from the previous version. Table 3.8 shows the results for this baseline technique. Columns 1-2 present the project names and average mutation scores of all versions for each subject. Columns 3-5 show error cells of directly copying mutation matrices from the previous version. Here, the average number of error cells is 5919, much larger compared with Ekstazi (381.93) and STARTS (356.23). Columns 6-8 show the error mutation scores, and there are three projects with maximum error mutation scores of even over than 10%. When further investigating the detailed reason, we found that there are massive code changes between such versions. For example, for project *compile-testing*, the maximum error mutation score is 16.77% due to the massive changes between version *5015d6* (with mutation score of 38.15%) and version *bf6de0* (with mutation score of 54.92%). In contrast, when we apply regression mutation testing here, the error mutation score is 0% using both Ekstazi and STARTS, since all tests are affected here.

Table 3.9: Efficiency of Ekstazi and STARTS

Project Name	Results			Ekstazi			STARTS		
	Tests	Mutants	Cells	Test Level	Mutant Level	Cell Level	Test Level	Mutant Level	Cell Level
HikariCP	1922.31	5523	672782	80.23%	78.84%	74.06%	75.41%	77.86%	71.93%
commons-io	99.00	10435	791820	39.04%	80.48%	79.24%	38.56%	79.37%	78.02%
OpenTripPlanner	138.55	12679	374177	81.92%	96.93%	88.24%	53.01%	95.14%	81.29%
commons-functor	156.51	7775	105144	93.23%	99.81%	98.76%	90.62%	99.63%	97.67%
commons-lang	142.06	37353	7274165	93.08%	91.27%	97.10%	80.97%	83.79%	93.92%
commons-net	42.78	14826	1017258	95.92%	98.96%	98.65%	95.83%	98.92%	98.56%
commons-text	42.20	7518	1281577	94.94%	92.89%	98.03%	95.12%	93.21%	98.11%
commons-validator	69.04	6680	2858829	96.92%	94.97%	98.79%	96.46%	94.42%	98.79%
compile-testing	9.57	3137	677448	75.60%	80.29%	93.24%	75.42%	80.04%	93.07%
invokebinder	3.07	1546	98889	63.22%	85.04%	91.77%	64.65%	87.22%	92.96%
logstash-logback-encoder	40.76	4068	698856	93.14%	92.76%	97.14%	94.44%	92.37%	97.04%
commons-codec	51.65	10533	2762729	97.46%	98.49%	99.58%	96.46%	97.69%	99.40%
commons-dbutils	26.28	2037	1309638	90.68%	91.36%	99.02%	86.72%	89.41%	98.63%
commons-scxml	353.22	9276	8911911	27.90%	61.70%	81.41%	17.41%	59.72%	79.94%
commons-csv	14.18	1336	352518	88.23%	84.55%	95.38%	85.35%	84.18%	95.01%
commons-jexl	43.00	33077	17435853	21.79%	61.70%	80.29%	17.75%	61.25%	79.73%
la4j	39.00	13561	3703834	49.64%	54.55%	79.95%	37.15%	49.43%	77.01%
commons-cli	24.48	2767	1480348	92.03%	90.36%	97.78%	90.09%	89.03%	97.25%
commons-math	436.48	113838	47057392	94.25%	91.99%	96.83%	85.28%	86.53%	93.80%
asterisk-java	57.00	18774	716054	96.78%	97.91%	96.18%	89.41%	96.54%	93.93%
Average	185.56	15837	4979061	78.30%	86.24%	92.07%	73.31%	84.79%	90.80%

3.3.2 RQ2: how do state-of-the-art static and dynamic RTS techniques perform in terms of regression mutation testing efficiency?

Table 3.9 shows the time savings achieved by Ekstazi and STARTS during regression mutation testing. In the table, Column 1 lists all the studied projects. Columns 2-4 present the average number of tests, mutants, and mutant-test cells executed by the original full mutation testing. Columns 5-7 present the test, mutant, and cell level reduction when using Ekstazi. Similarly, Columns 8-10 present the test, mutant and cell level reduction when using STARTS. In terms of test-level reduction, Ekstazi and STARTS reduce the number of tests by 78.30% and 73.31%, respectively, indicating the effectiveness of both STARTS and Ekstazi in test selection. We also use this metric to cross validate our execution of STARTS and Ekstazi with prior RTS study, and find that our numbers are consistent with prior RTS work [98]. In terms of mutant-level reduction, Ekstazi and STARTS reduce the number of executed mutants by 86.24% and 84.79%, respectively. The difference is smaller than that of the test-level reduction. In terms of the most precise cell-level reduction, Ekstazi and STARTS reduce the number of mutant-test cell executions by 92.07% and 90.80%, respectively. Note that the cell-level reduction values are much higher than the test or mutant level reduction values. The reason is that for any unreduced mutant or test, there can still be mutant-test cells that can be reduced by RTS. In summary, both Ekstazi and STARTS can greatly reduce mutation testing costs.

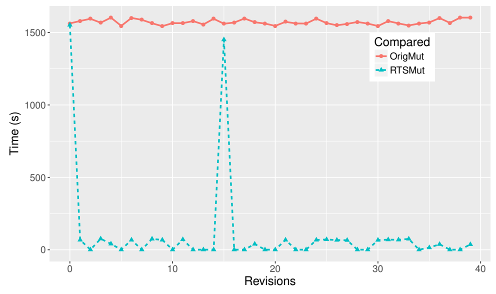


Figure 3.1: `asterisk-java`

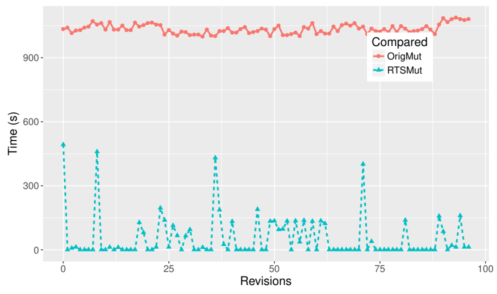


Figure 3.2: `commons-validator`

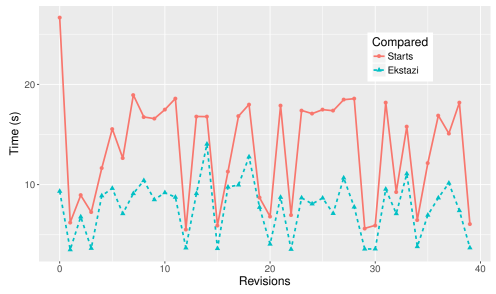


Figure 3.3: `asterisk-java`

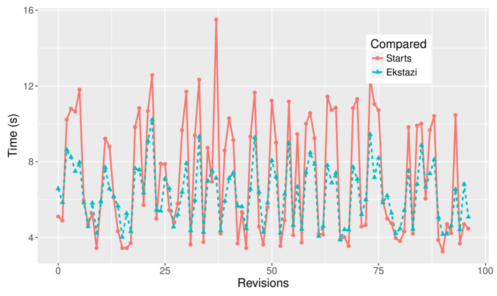


Figure 3.4: `commons-validator`

We further investigate the actual time savings achieved by state-of-the-art file-level RTS. Since both Ekstazi and STARTS achieved similar reduction, here we only present the actual time savings for Ekstazi on two example projects, `asterisk-java` and `commons-validator`. The experimental results on the other projects show similar trends. Figures 3.1 and 3.2 present the mutation testing time costs before and after using RTS. In each figure, the x-axis presents the number of revisions studied for the project, the y-axis presents the mutation testing time in seconds, the solid and dashed lines present the actual time when applying original mutation testing and Ekstazi-based mutation testing, respectively. From the figures, we can observe that Ekstazi-based regression mutation testing can significantly speed up mutation testing. For example, the original mutation testing costs 1035.22 seconds on average for

Table 3.10: Effectiveness and Efficiency of Faulttracer

Project Name	Efficiency			Effectiveness					
	Test Level	Cell Level	Mutant Level	Error Cells			Error MS		
				min	max	avg	min	max	avg
HikariCP	80.33%	78.74%	74.09%	0	8	15.13	0.0000%	1.61%	0.0510%
commons-io	42.46%	81.67%	80.55%	0	9	395.08	0.0000%	0.07%	0.0052%
OpenTripPlanner	85.99%	97.39%	90.07%	0	9	8.69	0.0000%	0.14%	0.0115%
commons-functor	93.37%	99.81%	98.76%	0	1	0.36	0.0000%	0.01%	0.0024%
commons-lang	96.99%	95.97%	98.79%	0	87	403.97	0.0000%	0.07%	0.0254%
commons-net	97.01%	99.18%	98.96%	0	99	119.93	0.0000%	0.03%	0.0048%
commons-text	95.45%	92.89%	98.05%	0	97	19.47	0.0000%	0.08%	0.0106%
commons-validator	98.24%	96.35%	99.32%	0	93	187.07	0.0000%	0.00%	0.0000%
compile-testing	91.30%	92.58%	97.60%	0	98	117.22	0.0000%	0.14%	0.0038%
invokebinder	70.29%	87.91%	93.34%	0	677	14.6	0.0000%	0.05%	0.0005%
logstash-logback-encoder	94.87%	93.99%	97.78%	0	9	17.19	0.0000%	0.38%	0.1108%
commons-codec	99.23%	99.42%	99.88%	0	9	28.18	0.0000%	0.08%	0.0172%
commons-dbutils	93.89%	93.97%	99.21%	0	86	399.98	0.0000%	0.50%	0.0276%
commons-scxml	30.24%	62.33%	81.74%	0	9	401.53	0.0000%	0.47%	0.0774%
commons-csv	93.64%	89.84%	97.79%	0	9	124.45	0.0000%	0.75%	0.1550%
commons-jexl	37.03%	63.72%	82.97%	1217	8459	1728.55	0.0000%	0.77%	0.1792%
la4j	74.03%	67.81%	87.03%	1080	914	563.03	0.0000%	0.10%	0.0111%
commons-cli	94.65%	92.62%	98.44%	0	962	97.61	0.0000%	0.00%	0.0000%
commons-math	97.50%	95.80%	98.31%	1084	8740	5123.69	0.0523%	0.89%	0.2944%
asterisk-java	99.41%	99.75%	99.64%	0	9	4.03	0.0000%	0.00%	0.0000%
Average	83.30%	89.09%	93.62%	169	1019	448.49	0.0026%	0.31%	0.0494%

`asterisk-java`, while Ekstazi-based mutation testing only costs 54.37 seconds, indicating a reduction of 94.75%. Similarly, the original mutation testing costs 1372.35 seconds on average for `commons-validator`, while Ekstazi-based mutation testing only costs 48.50 seconds, indicating a reduction of 96.47%.

In addition, we also investigate the RTS overhead. Figures 3.3 and 3.4 show the Ekstazi and STARTS overhead (i.e., end-to-end time including RTS analysis, selected test execution, and dependency collection) for `asterisk-java` and `commons-validator`. From the figure, we can observe that the cost of both static and dynamic RTS are negligible compared to the mutation testing time, e.g., the average Ekstazi and STARTS overhead for `asterisk-java` is only 7.71 seconds and 13.62 seconds, respectively.

In summary, file-level RTS techniques can significantly speed up mutation testing with negligible overhead.

Table 3.11: Effectiveness of Partial Regression Mutation

Project Name	Mutation Score	Ekstazi						STARTS					
		Error Cells			Error Mutation Score			Error Cells			Error Mutation Score		
		min	max	avg	min	max	avg	min	max	avg	min	max	avg
HikariCP	23.24%	0	104	3.56	0.0000%	1.61%	0.0418%	0	248	3.15	0.0000%	0.51%	0.0092%
commons-io	32.03%	0	17	3.26	0.0000%	0.14%	0.0071%	0	17	2.03	0.0000%	0.14%	0.0151%
OpenTripPlanner	6.89%	0	18	1.39	0.0000%	0.14%	0.0109%	0	6	0.47	0.0000%	0.05%	0.0037%
commons-functor	6.77%	0	1	0.18	0.0000%	0.01%	0.0024%	0	1	0.18	0.0000%	0.01%	0.0024%
commons-lang	78.75%	0	138	12.28	0.0000%	0.07%	0.0238%	0	27	9.64	0.0000%	0.07%	0.0206%
commons-net	23.16%	0	24	0.97	0.0000%	0.03%	0.0048%	0	24	0.97	0.0000%	0.03%	0.0048%
commons-text	74.08%	0	110	2.21	0.0000%	1.43%	0.0276%	0	110	2.21	0.0000%	1.43%	0.0276%
commons-validator	67.73%	0	2	0.44	0.0000%	0.00%	0.0000%	0	18	0.84	0.0000%	0.00%	0.0000%
compile-testing	56.22%	0	656	10.88	0.0000%	0.14%	0.0038%	0	656	10.88	0.0000%	0.14%	0.0038%
invokebinder	42.65%	0	1	0.01	0.0000%	0.05%	0.0005%	0	1	0.01	0.0000%	0.05%	0.0005%
logstash-logback-encoder	54.24%	0	100	6.46	0.0000%	0.62%	0.1160%	0	22	5.8	0.0000%	0.38%	0.1108%
commons-codec	77.90%	0	84	3.08	0.0000%	0.08%	0.0170%	0	84	3.08	0.0000%	0.08%	0.0170%
commons-dbutils	46.23%	0	258	12.22	0.0000%	0.50%	0.0276%	0	258	12.22	0.0000%	0.50%	0.0276%
commons-scxml	44.50%	0	349	121.25	0.0000%	0.96%	0.1779%	0	200	41.78	0.0000%	0.96%	0.1546%
commons-csv	71.87%	0	17	2.8	0.0000%	0.75%	0.1456%	0	17	2.66	0.0000%	0.75%	0.1405%
commons-jexl	46.10%	0	303	43.61	0.0000%	0.63%	0.0960%	0	295	32.34	0.0000%	0.63%	0.0759%
la4j	57.62%	0	26	1.1	0.0000%	0.02%	0.0030%	0	2	0.1	0.0000%	0.01%	0.0007%
commons-cli	84.43%	0	188	2.21	0.0000%	0.00%	0.0000%	0	188	2.21	0.0000%	0.00%	0.0000%
commons-math	71.47%	84	1471	422.46	0.0523%	0.57%	0.2820%	3	1471	380.32	0.0026%	0.57%	0.2572%
asterisk-java	17.88%	0	0	0.0	0.0000%	0.00%	0.0000%	0	0	0.0	0.0000%	0.00%	0.0000%
Average	49.19%	4.2	193.35	32.52	0.0026%	0.39%	0.0494%	0.15	182.25	25.54	0.0001%	0.32%	0.0436%

3.3.3 RQ3: how do different test dependency granularities impact the effectiveness and efficiency of RTS-based regression mutation testing?

To further study the impact of RTS with different granularity, we further apply FaultTracer, state-of-the-art dynamic method-level RTS, for regression mutation testing. We present the effectiveness and efficiency of FaultTracer-based RTS in Table 3.10. In the table, Column 1 lists all the projects. Columns 2-4 present the test, mutant, and cell level reductions. The remaining columns present the effectiveness metrics including both error cells and error mutation scores. From the table, we can observe that although FaultTracer has a higher test-level reduction (83.30%) than Ekstazi and STARTS (consistent with prior work [7]), the cell-level reduction is actually quite close to that of Ekstazi and STARTS. This observation demonstrates that finer-grained RTS does not provide clear benefits in efficiency. Furthermore, FaultTracer incurs more severe mutation testing precision issues. For example, on average, FaultTracer incurs 448.49 error cells while Ekstazi/STARTS only incurs 381.93/356.45 error cells. The reason is that a test whose method-level dependencies do not touch code changes may very likely be diverged to cover other changed method-level entities (whereas it is harder to diverge a test execution to execute other changed file-level entities). Therefore, finer-grained RTS brings more severe mutation testing precision issues without clearly improving mutation testing efficiency.

3.3.4 RQ4: how do state-of-the-art static and dynamic RTS techniques perform in partial regression mutation testing?

In this RQ, we further investigate the effectiveness of regression mutation testing under the partial mutation testing scenario. In Table 3.11, Columns 1 and 2 show the project names and the average mutation scores. They are the same as Table 3.7. Columns 3-5 present the error cells and Columns 6-8 present the error mutation scores when using Ekstazi. Similarly, Columns 9-11 and Columns 12-14 present the error cells and error mutation scores under STARTS. Based on the table, we have the following observations. First, the average error cells are 32.52 and 25.55 when using Ekstazi and STARTS. The average error cells are much fewer than those in the full regression mutation testing scenario (381.93 and 356.23 for Ekstazi and STARTS). Note that this is not because it is more accurate, but because the partial mutation testing scenario at most only has one killed test for each mutant, resulting in a much smaller total number of mutation cells than full mutation testing. Second, the average error mutation scores are only slightly higher than those in the full regression mutation testing scenario, e.g., 0.0494% and 0.0436% compared to 0.0423% and 0.0364% when using Ekstazi and STARTS, respectively. The slightly more inaccurate results here is because partial mutation only collects one killing test for each mutant, making imprecise results copied from pior versions have large impact on mutation scores. Note that overall RTS-based regression mutation testing still performs rather precisely in the partial scenario, demonstrating its effectiveness for both scenarios.

3.4 Threats to Validity

3.4.1 Internal

To reduce the threats to internal validity, we use state-of-the-art mature tools/techniques in our study. For example, we choose Ekstazi and STARTS to represent file-level dynamic and static RTS, and FaultTracer to represent dynamic method-level RTS. We also use state-of-the-art PIT mutation testing tool with all its 16 mutation operators. Furthermore, we also cross-validate our results with prior RTS or mutation studies.

3.4.2 External

Our experimental results might not generalize, since the projects used in our study are just a subset of all software systems and may not be representative. To reduce the threats, we used 1513 revisions of 20 real-world GitHub Java projects varying in size, application domain, number of tests, and running time. Also, all the used projects are single-module Maven projects for the ease of experimentation, and the results might be different for multi-module Maven projects. However, to our knowledge, this study already represent the largest scale study in the mutation testing literature.

3.4.3 Construct

Construct validity is mainly concerned with whether the used measurements are well designed and suitable for our study. To reduce this threat, we apply widely used effectiveness and efficiency measurements for mutation testing. For effectiveness, we use both the detailed error cell metric, and the error mutation score metric, both of which have been used in prior mutation testing work [182, 188]. For efficiency, we study the actual time reduction, as well as the reductions at the test, mutant, and cell levels.

CHAPTER 4
TAMING BEHAVIORAL BACKWARD INCOMPATIBILITIES VIA CROSS-PROJECT TESTING AND ANALYSIS

In Chapter 3, we have presented our approach to speed up mutation testing when software systems evolve. However, there are many other problems in the software testing field that are very time consuming and need to be made more efficient. In this chapter, we apply our faster revision testing in BBI bug detection to speed up the process of software testing.

As software products become larger and more complicated, library code plays an important role in almost any software. For example, while the sample Android app "Hello World" contains only several lines of source code, when it is executed on an Android mobile phone, it actually invokes libraries from the Android Software Development Kit (SDK), Java Development Kit (JDK), as well as the underlying Linux system. Third-party libraries such as Apache [22] and Square [21] libraries are also widely used in both open source and commercial software projects. The prevalent usage of software libraries has significantly reduced the software development costs and improved software quality.

At the same time, the asynchronous upgrades of software libraries and client software often result in incompatibilities between different library versions and client software. As techniques of computation evolve faster and faster, libraries are also upgraded more frequently, so do the occurrences of software incompatibilities. For example, Google releases a new major version of Android averagely every 11 months. After each major release, an outbreak of incompatibility-related bug reports will occur in GitHub, so do the version-upgrade-related negative reviews in the Google Play Market [116].

To avoid incompatibilities, for decades, "backward compatibility" has been well known as a major requirement in the upgrades of software libraries. However, in reality, "backward compatibility" is seldom fully achieved, even in widely used libraries. Some early research efforts (e.g., Chow and Notkin [53], Balaban et al. [40], and Dig and Johnson [60]) have confirmed the prevalence of backward incompatibility between two consecutive releases of software libraries. More recently, Cossette and Walker [56] identified 334 signature-level backward incompatibilities in 16 consecutive version pairs from 3 popular Java libraries: struts [13], log4j [20], and jDOM [19]. McDonnell et al. [116] identified 2,051 changes on method signatures in 13 consecutive Android API level pairs from API level 2-3 to API level 14-15. These studies all show that backward incompatibilities are prevalent. Furthermore, a recent study [124] found averagely over 12 test errors / failures from each version pair when performing cross-version testing on 68 consecutive version pairs of 15 popular Java libraries. This fact shows that, on top of signature-level backward incompatibilities, behavioral backward incompatibilities that may cause runtime errors instead of compilation errors are also prevalent.

Library incompatibilities may result in runtime failures both during the software development phase and after the software distribution. If the upgraded library is statically packaged in the client software product, the client developers may face some test failures when they try to incorporate the new release of the library. Thus they must perform extra changes and bug fixes if they want to take advantage of the new release of the library. In such a case, client developers may not be affected because they can still build the software product with the earlier library version. The case becomes worse when the upgraded library belongs to the runtime environment (e.g., operating system libraries, Java runtime libraries, platform libraries for plug-ins such as Chrome/Firefox/Eclipse libraries). In such cases, a software user may simply perform a system/platform update (the user may even not notice it if she turns on automatic updates) during the night, and suddenly find one or more software applications no longer working next morning. For example, Windows Vista is considered to be not very successful, and its failure has been largely ascribed to its backward incompatibility with

Windows XP [1]. More recently, an upgrade of Android platform from 4.4 to 5.0 broke SougouInput, the most popular Chinese-input software with more than 200 million users [2]. Users could not input any Chinese character after they upgraded to Android 5.0, until a patch was released 4 days later.

This paper proposes to apply *cross-project* testing and analysis to overcome the challenges in BBI detection with the following two insights. *First, the large number of open source client software projects residing in open software repositories can serve as a natural knowledge base of common usage scenario and expected semantics of software library APIs. Second, it is difficult for natural language documents (e.g., release notes) to achieve comprehensiveness and preciseness in describing semantic changes of library APIs. In contrast, code (including library and client code, source and test code) can be better media to transfer knowledge from the library side to the client side.* In particular, to avoid BBI-related software runtime failures, to accelerate software upgrading process, and to reduce developers' effort in software migration, we propose DeBBI to detect BBIs on library side. Simple cross-version regression testing with built-in library test code may miss a lot of BBIs. For better detection of BBIs, DeBBI leverages the large number of existing client software projects in open software repositories, and performs large-scale testing on these projects with their built-in test code on the newer library version. Such largely expanded test suites may incur high costs. Therefore, we propose to transform the problem of cross-project BBI detection into a traditional information retrieval (IR) problem. More specifically, we treat the library-side API upgrades as the query, and the project-side usage of the library APIs as the document collection. Then, the projects with more intensively upgraded API uses will be prioritized for early execution to detect potential BBIs faster. Also, different projects may share similar API uses and thus detect similar BBIs. Thus, we further consider the diversity between client projects using the diversified Maximal Marginal Relevance (MMR) technique [48]. Finally, for each client project, we also optimize test executions by skipping the tests that may not touch the upgraded APIs. The paper makes the following contributions:

- **Idea.** We propose to solve the BBI detection problem via *cross-project* testing and analysis, and further transform the problem into a traditional IR problem.
- **Implementation.** We implement the proposed approach for testing library BBIs based on the ASM bytecode analysis framework [14] and the Indri IR framework [16].
- **Optimization.** We further propose to use MMR to consider the diversity of different client projects, and also extend traditional static regression test selection to the cross-project scenario to automatically skip the tests useless for BBI detection.
- **Study.** We present an extensive study on testing JDK and other popular 3rd-party library (such as Apache libraries) upgrades using tens of thousands of GitHub client Java projects. The experimental results show that DeBBI can reduce the end-to-end testing time for detecting the first and average BBI clusters by 99.1% and 70.8% for JDK, and detect 97 real BBI bugs (19 has been confirmed as previously unknown bugs).

4.1 Approach

In this section, we first present the overview of our DeBBI approach (Section 4.1.1). Then, we illustrate how to apply IR techniques for efficient and effective BBI detection (Section 4.1.2). Finally, we show how to extend traditional Regression Test Selection (RTS) to the cross-project setting to further speed up DeBBI (Section 4.1.3).

4.1.1 Overview

Our DeBBI is a general approach for taming BBIs via cross-project testing, and can be applied to any library, including Android Software Development Kit (SDK) [12], Java Development

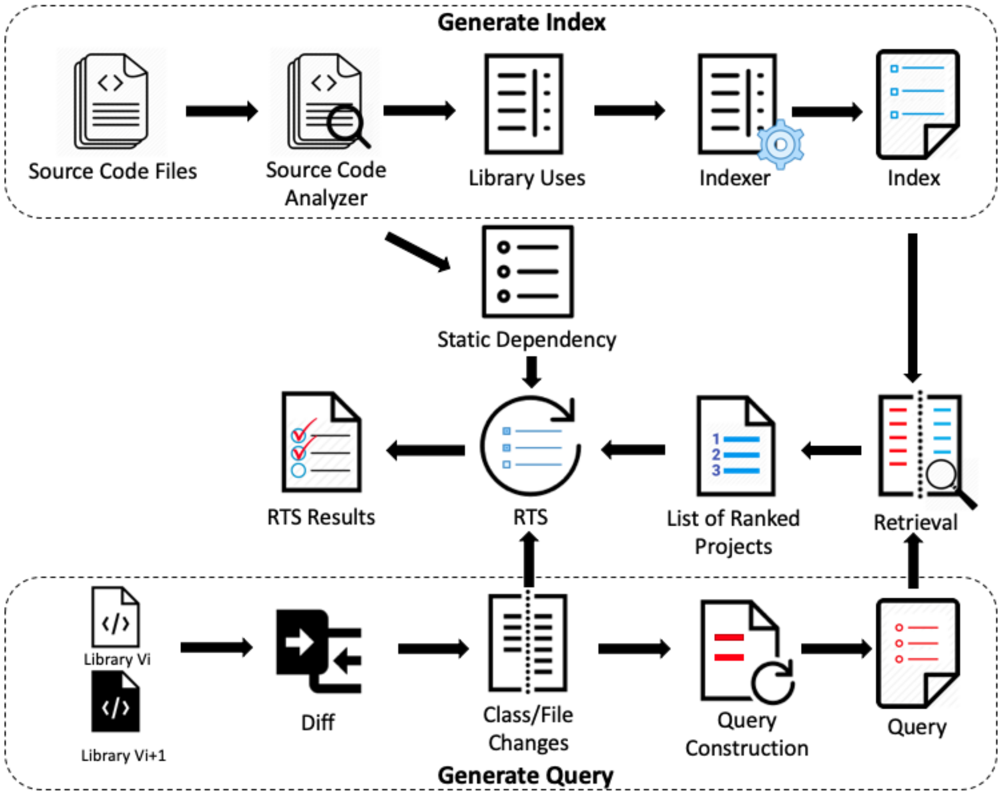


Figure 4.1: DeBBI structure

Kit (JDK) [17], and third-party libraries such as Apache Software [22]. Figure 4.1 shows the overall architecture of our DeBBI. DeBBI takes two versions of the library under test and a set of client projects that directly use the library as input to find BBIs. DeBBI first extracts the changes (e.g., file changes) among the two library versions via static analysis. They are considered as queries in our IR model. Meanwhile, DeBBI preprocesses the source code for all the client projects to obtain the library APIs used by each project, and uses that to serve as the document for each project during IR. Then, DeBBI queries the library changes against the source code for all the client projects, so that the client projects accessing more changed APIs are tested earlier to detect BBIs faster.

Following prior work [112, 114, 154, 191], we performed stop word removal [78], stemming [138] for the IR document preparation. Note that we use all Java key words as our stop word since they are common for all Java projects. For each client project, we consider the class/file-level dependencies on the library under test as the document contents. For each class/file, we split its fully-qualified name into different words in the document or query. For example, we split `java.lang.String` into `java`, `lang` and `String`. These three words are all fed into our document or query. To ensure DeBBI effectiveness and efficiency, we further explore various IR models in this work, including traditional and topic-model-based IR models (Details shown in Section 4.1.2). Furthermore, the client projects ranked high in the prioritization results may reveal similar or even the same BBIs. Therefore, we further consider the diversity of the IR results to detect *different* unique BBIs faster. To this end, we further use the Maximal Marginal Relevance (MMR) algorithm [48] to rank client projects with diverse library API uses.

IR models can help greatly reduce the number of client projects for finding BBIs. However, for each client project, all its tests are still executed. Therefore, in Section 4.1.3, we further use static analysis to compute the library APIs reachable from each test, and then compute the subset of tests which can potentially access changed library APIs as *affected tests*. In this way, for each client project, we only execute the affected tests to further speed up BBI detection.

4.1.2 DeBBI via Information Retrieval

Various IR models have been applied to solve software engineering problems, such as the Vector Space Model (VSM) [156], Latent Semantic Indexing (LSI) [94], and Latent Dirichlet Allocation (LDA) [44]. In theory, any IR model can be applied to DeBBI. In this work, we mainly consider two widely used IR models, VSM and LDA, due to their effectiveness [97, 174]. For each model, we studied state-of-the-art variants for effective BBI detection. Furthermore, for each studied variant, we further apply the Maximal Marginal Relevance (MMR) algorithm [48] to rank client projects with diverse library API uses.

Vector Space Model

Vector Space Model (VSM) [156] is an algebraic model for representing text documents and queries as vectors of indexed terms. TF.IDF (short for Term Frequency-Inverse Document Frequency) is a numerical statistic widely used to reflect word importance for a document under VSM. To date, TF.IDF and its variants (e.g., state-of-the-art Okapi BM25 [148]) have been widely recognized as robust and effective IR models [147]. Therefore, it has been widely studied and used in both IR and software engineering areas [133, 158, 171, 181]. Formally, assume that each document and query are represented by a term frequency vector $\vec{d}$ and $\vec{q}$ respectively, and n is the total number of terms or the size of vocabulary:

$$\vec{d} = (x_1, x_2, \ldots, x_n) \tag{4.1}$$

$$\vec{q} = (y_1, y_2, \ldots, y_n) \tag{4.2}$$

Element x_i and y_i are the frequency of term t_i in document $\vec{d}$ and query $\vec{q}$ respectively. Generally, query and document terms are weighted not just by their raw frequencies. There is a heuristic TF.IDF weighting formula to weight query and document term frequency (TF). Also, the inverse document frequency (IDF) is used to increase the weight of terms with low frequencies in the document and diminish the weight of terms which have high frequencies. Weighted vectors for $\vec{d}$ and $\vec{q}$ are computed as:

$$\vec{d_w} = (tf_d(x_1)idf(t_1), tf_d(x_2)idf(t_2), \ldots, tf_d(x_n)idf(t_n)) \tag{4.3}$$

$$\vec{q_w} = (tf_d(y_1)idf(t_1), tf_d(y_2)idf(t_2), \ldots, tf_d(y_n)idf(t_n)) \tag{4.4}$$

Given a set D of source files for the client projects considered by DeBBI, the simplest and classic TF formulation just uses the raw count of each term in the document, i.e., the number of times that term t occurs in a document, which is given by $f_{t,d}$. Similarly, one simplest way to calculate IDF is given by $idf(t) = \log \frac{N}{n_t}$, where n_t is the number of documents with term t and N is the total number of documents in document collection D. Thus, one of the simplest ways to get TF.IDF score is to just multiply $f_{t,d}$ and $\log \frac{N}{n_t}$ to get term t's score in document $\vec{d}$, and then compute the vector similarity with query $\vec{q}$ to get document $\vec{d}$'s priority.

As we mentioned before, various TF.IDF variants have been proposed in practice. In this work, we use the Indri [16] framework, which includes various advanced algorithms to achieve more accurate models. The Indri's TF.IDF variant is based on Okapi BM25, which is a probabilistic retrieval framework model initially developed by Robertson et al. [148]. As to avoid division by zero, when a particular term appears in all documents, the IDF value here is: $idf(t) = \log \frac{N+1}{n_t+0.5}$. Meanwhile, the TF value is:

$$tf_d(x) = \frac{k_1 x}{x + k_1(1 - b + b\frac{len_d}{len_D})} \tag{4.5}$$

There are two tuning parameters $k1$ and b. $k1$ is used to calibrate document term frequency scaling. When $k1$ is just a small value, the term frequency value will quickly saturate; on the contrary, a large $k1$ value corresponds to using raw term frequency. $b(0 \leq b \leq 1)$ is used to determine the scaling by document length. When value b is 1, it corresponds to fully scaling the term weight by the document length, while $b = 0$ corresponds to no length scaling. Finally, len_d and len_D represent the current document length and average document length for the entire document collection, respectively.

Meanwhile, for the query's TF function, the length normalization is unnecessary because retrieval is applied with respect to a single fixed query. Therefore, we just set b as 0 here:

$$tf_q(y) = \frac{k_3 y}{x + k_3} \tag{4.6}$$

Thus, the similarity score of document $\vec{d}$ against query $\vec{q}$ is:

$$S(\vec{d}, \vec{q}) = \sum_{i=1}^{n} tf_d(x_i) tf_q(y_i) idf(t_i)^2 \tag{4.7}$$

There are various configurations that we can choose in the Indri framework. One of them is the basic TF.IDF variant using BM25TF term weighting. It sets k_3 as 1000 in the equation 6. The only two parameters left for tuning are $k1$ (for term weight) and b (for term weight). We directly use their default values, i.e., 1.2 and 0.75, respectively. Another variant is Okapi, which performs retrieval via Okapi scoring. There are three parameters $k1$ (for term weight), b (for term weight), and $k3$ (for query term weight) in the variant. The default value of them are 1.2, 0.75 and 7 respectively. We also use these default values in our experiment. In this work, we use both models and denote them as TF.IDF and Okapi, respectively.

Latent Dirichlet Allocation

Different from VSM that directly represents documents with indexed terms, LDA further implements topic modeling in the retrieval process and computes generative statistical models to split a set of documents into corresponding topics with certain probabilities. In this way, each document is represented by the set of relevant abstract topics rather than the raw indexed terms. In the software engineering literature, researchers have applied LDA to deal with bug localization [191], software categorization [170], or software repository analysis [166]. In those prior works, project source code is usually treated as LDA input documents. In contrast, in this work, DeBBI treats each client project's class-level dependency on the library under test as LDA input documents. Based on the input documents, LDA computes different topics for each of the client projects. The different topics indicate that there are different clusters of projects. When projects use very similar library APIs, they are assigned into similar topics.

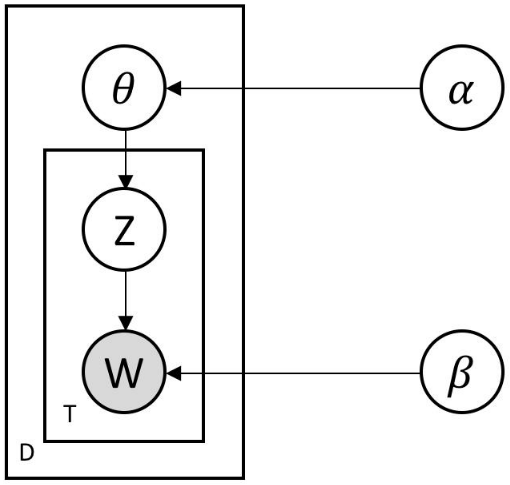


Figure 4.2: Graphical model for LDA

Figure 4.2 shows the graphical model of LDA. The outer box D represents the documents. The inner box T represents the repeated choice of topics and words in a document. The generative process of model can be described as follows:

(1) Choose $T \sim \text{Poisson}(\epsilon)$

(2) Choose a topic vector $\theta \sim \text{Dir}(\alpha)$ for document D

(3) For each of the T terms w_i:

    (a) Choose a topic $z_j \sim \text{Multinomial}(\theta_D)$

    (b) Choose a term w_i from $p(w_i|z_j, \beta)$

For here, α is a smoothing parameter for document-topic distributions, and β is a smoothing parameter for topic-term distributions. The multinomial probability function p is:

$$p(\theta, z, w|\alpha, \beta) = p(\theta|\alpha) \prod_{n=1}^{T} p(z_n|\theta) p(w_n|z_n, \beta) \tag{4.8}$$

In this way, given a set of client projects, we first generate a term-by-document matrix $\vec{M}$. Then we use w_{ij} to represent the weight of i_{th} term in the j_{th} document. Note that following prior work [79, 92], we take TF.IDF as our weighting function, which can give more importance to words with high frequency in the current document and appearing in a small number of documents. LDA further takes the $\vec{M}$ as input, and produces a topic-by-document matrix $\vec{R}$. For here, the probability that the j_{th} document belongs to the i_{th} topic is denoted by R_{ij} in this matrix. Because the number of topics is much smaller than the number of indexed terms in the corpus. LDA is mapping a high-dimensional space of documents into a low-dimensional space (represented using topics). The latent topics can be clustered by shared topics.

In the implementation, we apply the fast collapsed Gibbs sampling generative model [137] for LDA. The reason is that it is much faster and has the same accuracy compared against the standard LDA implementation [44]. There are the following parameters in the model which may affect its performance:

- t, which is the number of topics in the result. Follow the prior work [39], we set topic number as 10 in our experiment.
- n, which denotes the number of Gibbs iterations to train our model. And we set it as 10000 in the experiment following prior work [142].
- α, which influences the topic distributions per document. The topics will have a better smoothing effect when the α value is higher. We use the default value of 5.5.
- β, which influences the term's distribution per topic. The distribution of terms per topic will be more uniform with a higher β value. We use the default value of 0.01.

Maximal Marginal Relevance

Both the VSM and LDA techniques above will rank the most relevant client projects high in the list. However, the highly ranked projects may access similar library APIs and reveal the same BBIs repetitively. Therefore, in this work, we further consider the diversity among the search results to detect different unique BBIs faster. More specifically, we combine both VSM and LDA models with Maximal Marginal Relevance (MMR) [48] to solve this diversity issue to explore their performance. MMR has been widely studied in the IR community for diversified searching [71, 75, 89, 96]. Traditional IR models rank the retrieved documents in the descending order of relevance to the user's query. In contrast, MMR tries to measure relevance and novelty independently and consider them together via a linear combination to solve the diversity problem. For example, it maximizes marginal relevance in retrieval and summarization when a document is both relevant to the query and contains minimal similarity to the previously ranked documents. The MMR score equation can be formally defined as:

$$A_{rg} \max_{d_i \in D \setminus S} [\lambda(Sim_1(d_i, q) - (1-\lambda) \max_{d_j \in S} Sim_2(d_i, d_j))] \tag{4.9}$$

where D is the document collection (i.e., the set of considered client projects for testing a library using DeBBI) and q is the query (i.e., the changes among different library versions). S is the subset of documents which are already selected by IR. $D \setminus S$ is the set of not yet selected documents in D. Sim_1 and Sim_2 are the methods to measure similarity between documents and query. They can be the same or different. For here, we uniformly use BM25 [173] as our similarity calculation method. In the above definition, when parameter $\lambda = 1$, MMR gives us a standard relevance-ranked list. On the contrary, when $\lambda = 0$, MMR gives us a maximal diversity result. In addition, the sample information space is around the query when λ is a small number, whereas the larger value of λ will produce a result focusing on multiple potentially overlapping or reinforcing relevant documents. In our experiment we set λ as 0.5 which gives documents and queries the same weight.

4.1.3 Faster DeBBI via Testing Selection

Since the basic DeBBI only ranks client projects, all the tests within each tested projects still have to be executed. Therefore, we further extend DeBBI to reduce the number of test executions within each project. More specifically, we extend the traditional Regression Test selection (RTS) approach [150] to further enable even faster BBI detection. To date, various static and dynamic RTS techniques have been proposed in the literature [65, 70, 98, 160, 186]. In this work, we build DeBBI on top of state-of-the-art static RTS technique STARTS [98]. We chose STARTS since it has been demonstrated to be state-of-the-art static file-level RTS technique and can be competitive to state-of-the-art dynamic RTS technique Ekstazi [70]. Also, STARTS does not require prior dynamic execution information for each client project, which may not be available during BBI detection. STARTS is based on the traditional class firewall analysis firstly proposed by Leung et al. [93, 99]. To further consider the specific features of the Java programming language, STARTS performs class firewall analysis on the Intertype Relation Graph (IRG) defined by Orso et al. [131]. The following presents the formal definition:

Definition 4.1.1 (Intertype Relation Graph)**.** *The intertype relation graph of a given Java program can be formulated as a triple $\langle \mathcal{E}, \mathcal{N}_i, \mathcal{N}_u \rangle$. In the triple, N denotes the set of nodes representing all programs' classes or interfaces. $\mathcal{E}_i \subseteq \mathcal{N} \times \mathcal{N}$ denotes the set of inheritance edges. There exists an inheritance edge $\langle n_1, n_2 \rangle \in \mathcal{E}_i$ if type n_1 inherits from class n_2, or implements interface n_2. $\mathcal{E}_u \subseteq \mathcal{N} \times \mathcal{N}$ denotes the set of use edges. There exists an edge $\langle n_1, n_2 \rangle \in \mathcal{E}_u$ if type n_1 accesses any element of n_2, e.g., field references and method calls.*

There are two inputs for STARTS to select affected tests: (1) the set of changed files during software evolution, (2) the static dependency for each test computed based on the IRG graph, i.e., the files that can potentially be reached from each test based on IRG. Then, STARTS computes all files that can potentially reach the changed files within the class firewall, and all tests within the firewall will be selected for execution. Formally, the class firewall can be computed as:

Definition 4.1.2 (Class Firewall)**.** *The class firewall for a set of changed types $\tau \subseteq \mathcal{N}$ is computed over the IRG $\langle \mathcal{N}, \mathcal{E}_i, \mathcal{E}_u \rangle$ using as the transitive closure computation: $firewall(\tau) =$*

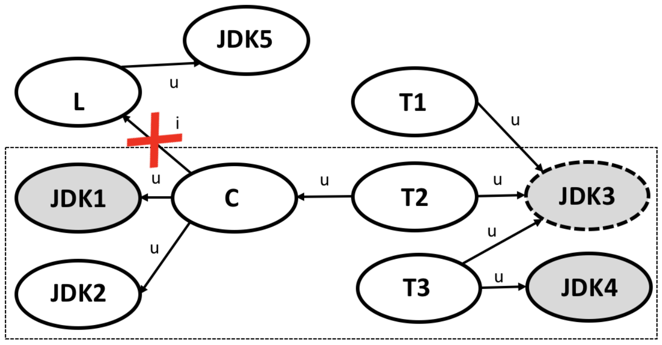


Figure 4.3: Example IRG

$\tau \circ \overline{\mathcal{E}}^*$, where $\circ$ is the relational product, * denotes the reflexive and transitive closure, and $\overline{\mathcal{E}}$ denotes the inverse of all use and inheritance edges, i.e., $(\mathcal{E}_i \cup \mathcal{E}_u)^{-1}$.

Note that the prior STARTS approach only analyzes the nodes within a project (ignoring all third-party and JDK libraries). On the contrary, in this work, we explicitly consider library changes, and aim to select the tests affected by library changes. Therefore, we augment the STARTS analysis to include library nodes. Note that (1) DeBBI only considers the nodes for the client projects and the library under test, and ignores all the other library nodes, and (2) DeBBI only considers the library nodes directly reachable from client projects. The reason is that the nodes for other libraries are not of interest, and the library nodes not directly reachable from the client projects may not have clear impact on the current project. For example, when applying DeBBI to detect JDK BBIs, we don't consider the third-party library dependencies and only collect the source code and test code JDK dependencies through jDeps [18]. Then we set the changed JDK library files as our code changes for test selection. Note that, we further filter out the top 200 most widely used JDK files, such as `java.lang.String` and `java.util.List`. The reason is that these files are almost used by all projects/tests and cannot help much in test selection. Note that we empirically validated that after filtering these JDK classes, our test selection is still safe, i.e., not missing any unique BBI.

Figure 4.3 illustrates how we adapt RTS for detecting BBIs for JDK. In the example IRG, the inheritance and use edges are marked with label "i" and "u". L denotes a third-party library node, which uses JDK node JDK_5; C is a client project node which inherits library L and uses JDK_1 and JDK_2. There are three tests T_1, T_2 and T_3 all using JDK_3. According to our approach, we do not consider the dependencies of third-party library, and thus JDK_5 will not be considered in our dependency result (pruned by red cross mark). In addition, we just consider one layer JDK dependency. For example, we only collect JDK dependencies of C, T_1, T_2 and T_3. We do not consider the further dependencies of JDK_1, JDK_2, JDK_3 and JDK_4. From the figure, T_2 uses client C and T_3 uses JDK_4, respectively. JDK_1, JDK_3 and JDK_4 are the changed JDK classes (marked with gray shadow). Note that JDK_3 is one of the 200 most commonly used JDK classes, and it will not be considered in JDK diff results as discussed before (marked with dashed oval). In this way, T_2 can potentially reach JDK_1 and T_3 is using changed class JDK_4. Thus, T_2 and T_3 are affected tests in our RTS technique, marked within the dashed area (i.e., our class firewall).

4.2 Experimental Setup

In this section, we first described our dataset for detecting JDK BBIs (Section 4.2.1), followed by our evaluation environment (Section 4.2.2), and evaluation metrics (Section 4.2.3).

4.2.1 Dataset

To construct the dataset for detecting JDK BBIs, we first collect all the most-forked Java projects with over 20 forks from the GitHub repository. It returns a collection of 8,481 unique Java projects. In these resulting projects, 4,928 of them support the Maven build system. Finally, we use all the 2,953 remaining projects can pass the build and test phases successfully under JDK 8 as the dataset for this study.

Table 4.1: Dataset summary

Description	Min	Max	Avg.
# Number of Java Files per Project	1	12979	130.37
# Number of Test Cases per Project	0	665028	329.68

Table 4.1 describes the dataset in more details. In particular, the number of Java source files in a project ranges from 1 to 12,979, and the number of test cases in a project ranges from 0 to 665,028. The average number of Java source files and the average number of test cases are 130.37 and 329.68, respectively. Since we would like to find BBI issues for different versions of JDK, the same dataset is applied to build and test with different JDK versions.

4.2.2 Experiment Settings

To perform our experiment, we need a set of confirmed JDK BBI bugs as ground truth. We use the dataset described in Section 4.2.1 to detect such confirmed BBI bugs. The intuition is that, we can confirm a BBI bug by checking whether it is fixed in the later versions of JDK. If a test case passes in JDK 8 but fails in JDK 9.0.0, then it reveals a BBI between JDK 8 and 9.0.0. However, we are not sure whether this BBI is an intended behavior change by JDK developers or a BBI bug. To confirm that such a BBI is a BBI bug, we further run the test case on 9.0.1, and if the BBI disappears, we confirm that the test failure in JDK 9 reveals a BBI bug. To categorize duplicated BBI bugs, we manually cluster all the reported BBIs caused by the same root issues to identify unique BBI bugs. In this way, we define every reported BBI as a *raw BBI bug* and every clustered BBI as a *unique BBI bug.* Note that we consider both raw and clustered bugs to better measure DeBBI effectiveness.

When performing the build and testing, we use Maven 3.3.9 to build and test each project. For the JDK version, we use JDK 8.0.161, 9.0.0 and 9.0.1. We use a computer with Intel(R) Xeon(R) CPU 2.60GHz with 528GB of Memory, and Ubuntu 16.04.3 LTS operating system.

4.2.3 Evaluation Metrics

We use each of the following three metrics to evaluate the number of projects tested, the number of test executions and amount of time taken to identify BBIs:

- **First:** This metric reports the number of client software projects tested, the number of tests executed, or time (in second) taken to identify the first BBI bug. This metric emphasizes fast detection of the first BBI, which is essential for the developers to start debugging earlier.
- **Average:** This metric is the average number of client software projects tested, tests executed, or average time taken to find each BBI. This metric emphasizes fast detection of BBIs in average cases.
- **Last:** Like the **First** metric, this metric reports the number of client software projects evaluated, the number of tests executed and time taken to identify the last BBI. This metric emphasizes fast detection of all BBIs.

4.3 Result Analysis

In this section, we seek to answer the following five research questions.

4.3.1 RQ1: Is DeBBI more effective than random project prioritization in identifying BBI issues?

To evaluate DeBBI on detecting BBIs for JDK, we compared the basic IR-based DeBBI with the Random technique, which randomly sorts client projects to identify BBIs. Also, the Random technique results are averaged over 5 runs to isolate the impact of random factors. We compared our results with the Random technique from three aspects: i) effectiveness in the number of tested client software projects, ii) effectiveness in the number of executed

Table 4.2: Effectiveness of the basic DeBBI

	Without Bug Clustering									**With Bug Clustering**								
	Client Software Projects			**Test Case**			**Execution Time(sec)**			**Client Software Projects**			**Test Case**			**Execution Time(sec)**		
	First	**Last**	**Average**	**First**	**Last**	**Average**	**First**	**Last**	**Average**	**First**	**Last**	**Average**	**First**	**Last**	**Average**	**First**	**Last**	**Average**
Random	63	2702	1607	1253	961050	776892	1494.97	109005.21	73360.5	63	2402	1663	1253	947219	758567	1494.97	103965.45	76463.23
TF.IDF	**3 (95.2%)**	2487 (8.0%)	**1322 (17.7%)**	**32 (97.4%)**	964053 (-0.3%)	399301 (48.6%)	**53.9 (96.4%)**	110727.5 (-1.6%)	77340.8 (-5.4%)	**3 (95.2%)**	1901 (20.9%)	1135 (31.7%)	**32 (97.4%)**	948943 (-0.2%)	413608 (45.5%)	**53.9 (96.4%)**	88637.3 (14.7%)	67582.5 (11.6%)
Okapi	5 (92.1%)	**2379 (12.0%)**	1375 (14.4%)	48 (96.2%)	962737 (-0.2%)	457375 (41.1%)	91 (93.9%)	109132.7 (-0.1%)	74956.7 (-2.2%)	5 (92.1%)	**1888 (21.4%)**	**982 (41.0%)**	48 (96.2%)	949122 (-0.2%)	241894 (68.1%)	91 (93.9%)	**87215.6 (16.1%)**	**60150.7 (21.3%)**
LDA	43 (31.7%)	2445 (9.5%)	1532 (4.7%)	573 (54.3%)	**727141 (24.3%)**	**167110 (78.5%)**	263.9 (82.3%)	**94113.1 (13.7%)**	**48290.1 (34.2%)**	43 (31.7%)	2332 (2.9%)	1747 (-5.1%)	573 (54.3%)	**711989 (24.8%)**	**108083 (85.8%)**	263.9 (82.3%)	90799.8 (12.7%)	64822.2 (15.2%)

tests, and iii) effectiveness in test execution time. For each aspect, we measure the **First**, **Average**, and **Last** metrics of both the Random and our IR-based techniques. The results are presented in Table 4.2. In the left half of the table, we present the **First**, **Last**, and **Average** values on client software projects, test executions, and execution time without bug clustering. The values in the bracket are the relative reduction for the corresponding metrics compared with the Random technique. The best technique for each metric has also been marked in gray.

We have following observations for the bugs without clustering: First, all IR-based techniques perform much better than the Random technique on the **First** values, with mostly 60% to 90% reduction on all three aspects. However, if we consider **Average** and **Last** values, the enhancement of IR-based techniques is not that significant, especially for execution time. This can be due to the lack of diversity in IR-based prioritization results. Second, there is none IR-based technique that outperforms all other techniques, but LDA is performing better (with 4.7% to 82.3% reduction) than Random technique on all values from all aspects.

As same BBI bugs can appear in multiple projects and test cases, we also performed BBI clustering to check how different techniques compare on identifying different *unique* BBI bugs. The right half of Table 4.2 shows the effectiveness of IR based techniques and Random technique on unique BBI bugs. The data presentation is the same as the left half. We have similar observations compared with left half of the table: IR-based techniques perform much better on **First** values, but not so good on **Last** and **Average** values. Furthermore, in general, IR-based techniques perform better than the Random technique on all values in test

Table 4.3: Effectiveness of DeBBI with MMR

	Without Bug Clustering									With Bug Clustering								
	Client Software Projects			Test Case			Execution Time(sec)			Client Software Projects			Test Case			Execution Time(sec)		
	First	**Last**	**Average**	**First**	**Last**	**Average**	**First**	**Last**	**Average**	**First**	**Last**	**Average**	**First**	**Last**	**Average**	**First**	**Last**	**Average**
Random	63	2702	1607	1253	961050	776892	1494.9	109005.2	73360.5	63	2402	1663	1253	947219	758567	1494.9	103965.5	76463.2
TF.IDF+MMR	28 (55.6%)	2404 (11.0%)	1306 (18.7%)	5791 (-362.2%)	961604 (-0.1%)	**515462 (33.7%)**	4104.5 (-174.6%)	109603.5 (-0.5%)	79052.8 (-7.8%)	28 (55.6%)	1591 (33.8%)	867 (47.9%)	5791 (-362.2%)	944968 (0.2%)	**369206 (51.3%)**	4104.5 (-174.6%)	85428.7 (17.8%)	59618.8 (22.0%)
Okapi+MMR	25 (60.3%)	2398 (11.3%)	1324 (17.6%)	5759 (-359.6%)	963338 (-0.2%)	559859 (27.9%)	4057.2 (-171.4%)	109540.7 (-0.5%)	81400.4 (-11.0%)	25 (60.3%)	1672 (30.4%)	878 (47.2%)	5759 (-359.6%)	949900 (-0.3%)	450257 (40.6%)	4057.2 (-171.4%)	86264.9 (17.0%)	59906.6 (21.7%)
LDA+MMR	**1 (98.4%)**	**2340 (13.4%)**	**1243 (22.7%)**	**1 (99.9%)**	**959254 (0.2%)**	759536 (2.2%)	**12.7 (99.1%)**	**105832.7 (2.9%)**	**55970.2 (23.7%)**	**1 (98.4%)**	**1029 (57.2%)**	**616 (63.0%)**	**1 (99.9%)**	**931735 (1.6%)**	553400 (27.0%)	**12.7 (99.1%)**	**42645.6 (59.0%)**	**26433.9 (65.4%)**

execution time for unique BBI bugs. The reason is that for unique BBI bugs DeBBI only need to find the first raw BBI bug in each cluster, making it easier for IR-based DeBBI to find unique BBI bugs faster.

4.3.2 RQ2: How does diversity resolution technique help improve the performance of DeBBI?

To check whether diversity enhancement techniques such as Maximal Marginal Relevance (MMR) can enhance IR-based project prioritization, we combine MMR with all IR-based techniques TF.IDF, Okapi and LDA. Table 4.3 shows the effectiveness of MMR-integrated IR-based techniques. From the table, we can see that although MMR is not very helpful on some IR techniques (TF.IDF and Okapi) in all aspects, it is able to enhance the LDA-based technique significantly. LDA+MMR outperforms all other techniques on almost all values from all aspects. Comparing with results in Table 4.2, we can see that MMR technique can enhance LDA-based technique on five of nine evaluated metrics without bug clustering and seven of nine metrics with bug clustering. In particular, when it comes to bug clustering, LDA+MMR is able to reduce 99.1%, 59.0%, and 65.4% of test execution time to detect the **First**, **Last**, and **Average** unique BBI bugs, which is a huge enhancement over the Random technique.

4.3.3 RQ3: Can we further boost DeBBI via extending traditional static Regression Test Selection (RTS)?

When a library gets updated, not all the tests from its client projects are affected by the library code changes. If we can remove such irrelevant test cases, we may further enhance

Table 4.4: Effectiveness of DeBBI with RTS

	Without Bug Clustering						**With Bug Clustering**					
	Test Case			**Execution Time(sec)**			**Test Case**			**Execution Time(sec)**		
	First	**Last**	**Average**	**First**	**Last**	**Average**	**First**	**Last**	**Average**	**First**	**Last**	**Average**
Random	1253	961050	776892	1494.9	109005.2	73360.5	1253	947219	758567	1494.9	103965.5	76463.2
Random+RTS	337 (73.1%)	27016 (97.2%)	14402 (98.1%)	1257.5 (15.9%)	71083.7 (34.8%)	40856.8 (44.3%)	337 (73.1%)	23985 (97.5%)	15039 (98.0%)	1257.5 (15.9%)	62587.8 (39.8%)	41810.2 (45.3%)
TF.IDF+RTS	6 (99.5%)	28013 (97.1%)	21564 (97.2%)	303.6 (79.7%)	74613.8 (31.6%)	52714.6 (28.1%)	6 (99.5%)	27021 (97.1%)	18918 (97.5%)	303.7 (79.7%)	67659.2 (34.9%)	46428.2 (39.3%)
TF.IDF+MMR+RTS	1474 (-17.6%)	27987 (97.1%)	23073 (97.0%)	3260.2 (-118.1%)	74642.2 (31.5%)	54315.2 (26.0%)	1474 (-17.6%)	26298 (97.2%)	18922 (97.5%)	3260.2 (-118.1%)	63198.1 (39.2%)	41769.4 (45.4%)
Okapi+RTS	2 (99.8%)	27719 (97.1%)	22009 (97.2%)	**82.9 (94.5%)**	98866.6 (9.3%)	76910.7 (-4.8%)	2 (99.8%)	26787 (97.2%)	17881 (97.6%)	278.3 (81.4%)	66228.2 (36.3%)	43050.3 (43.7%)
Okapi+MMR+RTS	739 (41.0%)	27996 (97.1%)	23457 (97.0%)	3038.8 (-103.3%)	74678.5 (31.5%)	55316.7 (24.6%)	739 (41.0%)	26698 (97.2%)	18636 (97.5%)	3038.8 (-103.3%)	64302.9 (- 38.1%)	42449.7 (44.5%)
LDA+RTS	210 (83.2%)	**9284 (99.0%)**	**4020 (99.5%)**	507.3 (66.1%)	**50285.1 (53.9%)**	**27010.3 (63.2%)**	210 (83.2%)	**7535 (99.2%)**	**4221 (99.4%)**	507.3 (66.1%)	46847.2 (54.9%)	31159.9 (59.2%)
LDA+MMR+RTS	**1 (99.9%)**	26287 (97.3%)	22274 (97.1%)	197 (86.8%)	69353.1 (36.4%)	46072.8 (37.2%)	**1 (99.9%)**	22692 (97.6%)	18003 (97.7%)	**12.7 (99.1%)**	**33241.9 (68.0%)**	**22300.2 (70.8%)**

the reduction on the number of test executions and execution time. Therefore, we further exclude the test cases that will not be affected by JDK code changes via RTS. The results of techniques with RTS combined are presented in Table 4.4, where the Random technique is used as the baseline for comparison. From the table, we can see that, with RTS combined, even Random+RTS also achieves good effectiveness (average execution time reduced from more than 70K seconds to about 41K seconds); meanwhile, DeBBI models tend to have even larger improvements. In addition, on detecting clustered unique BBI bugs, the LDA+MMR technique, which has achieved best effectiveness without RTS, still achieves significant enhancement over the Random technique when RTS is combined. Specifically, LDA+RTS can achieve 63.2% reduction on detecting raw BBI bugs and LDA+MMR+RTS can achieve 70.8% reduction on detecting unique BBI bugs compared with the Random technique on **Average** execution time. In other words, DeBBI can save 1017.1 hours to find all raw BBI bugs and 120.4 hours to find all unique BBI bugs.

In reality, detecting a new unique BBI bug is apparently more important than finding another instance of a known BBI bug. Therefore, we believe LDA+MMR+RTS is the best technique that we recommend to be used by default in reality. To make it more convenient to check the necessity of each used component (i.e., LDA, MMR, and RTS) compared to baseline techniques, we present the comparison among four selected techniques: Random technique, LDA, LDA+MMR, and LDA+MMR+RTS on clustered unique BBIs in Figures 4.4 to 4.6.

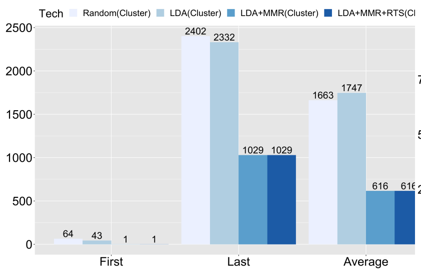


Figure 4.4: Client project execution

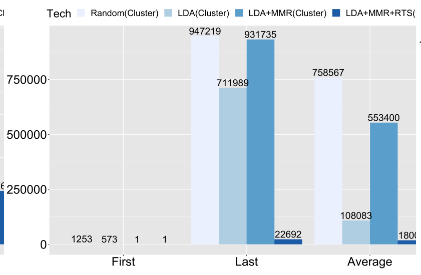


Figure 4.5: Test case execution

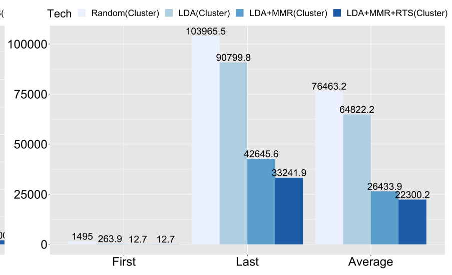


Figure 4.6: Test execution time

In particular, Figure 4.4 compares all four techniques on their **First**, **Last**, and **Average** values on the number of client project executions. Figure 4.5 and Figure 4.6 present similar comparison on the number of test executions and execution time. As shown in Figure 4.4, for prioritization of the client software projects, since RTS does not optimize project selection, LDA+MMR and LDA+MMR+RTS show same effectiveness. However, if we compare LDA+MMR+RTS with Random approach, it shows 98.4% 57.2% and 63.0% reduction on **First**, **Last**, and **Average** values respectively over the Random technique. As shown in Figure 4.5, from the aspect of test cases, LDA+MMR+RTS achieves 99.9%, 97.6%, and 97.6% for **First**, **Last**, and **Average** values over the Random technique. As shown in Figure 4.6, from the aspect of execution time, LDA+MMR+RTS achieves 99.1%, 68.0%, and 70.8% reduction **First**, **Last**, and **Average** values over the Random technique.

4.3.4 RQ4: How does DeBBI perform in case of parallel execution?

We further utilized the `multiprocessing` package of Python for parallel project execution. We used Python `Pool` to control the different processes to start or join in the main process and used `Manager` and `Queue` to control the shared resource between processes. In our experiments, the ranked project list from our IR-based result is the shared resource. Sub-processes try to get the project from queue and run it. As soon as one process finishes execution, it starts to get the next one to run. Here, we use 5 sub processes in our experiment to evaluate our technique. Table 4.5 shows the results of DeBBI with and without bug clustering during parallel execution. The left part is the execution time without bug clustering and right part is the execution time with bug clustering. Column 1 list all techniques. Columns 2-7 list **First**, **Last** and **Average** value of execution time to find raw BBI bugs and unique BBI bugs respectively. We use the Random technique with multiprocessing as our baseline technique. From the results, we can see that TF.IDF with MMR, Okapi and LDA with MMR all can find first raw BBI bug and unique BBI bug in 12.7 seconds with the 84.7% reduction compared to Random. LDA has the best performance in **Last** and **Average** with 11.8 % and 38.4 % reduction without bug clustering. Meanwhile, TF.IDF with MMR has the best performance in **Last** and **Average** with 80.9 % and 63.2 % reduction with bug clustering.

Table 4.6 shows the results when combining our IR-based techniques with RTS during parallel project execution. We still use the Random technique with multiprocessing as our baseline to check the results. From the results, all techniques combined with RTS can have a huge enhancement in **Last** and **Average** value of execution time. The reason LDA+RTS is better than Random in **First** is that RTS does not have too much help here. Random and most techniques can find first bug fast without RTS and executing RTS needs extra overhead[1]. Thus, the performance of **First** is not very good here. However, LDA+MMR+RTS is able to have 71.4 % and 83.1 % reduction in **Last** without and with bug clustering. LDA+RTS can have 64.4 % and 60.8 % average time reduction to find raw BBI bugs and unique BBI bugs. To sum up, LDA+MMR+RTS is still one of the most effective techniques in the setting of parallel project execution. It can save 129.3 hours to find all raw BBI bugs and 9.9 hours to find all unique BBI bugs compared to the Random technique with parallel execution.

[1]Note that all the RTS overhead costs, including computing dependencies and performing RTS analysis, are considered in our DeBBI time measurement.

Table 4.5: DeBBI for parallel project execution

	Without Bug Clustering			With Bug Clustering		
	Execution Time(sec)			Execution Time(sec)		
	First	**Last**	**Average**	**First**	**Last**	**Average**
Random	83.5	53122.1	15026.5	83.5	51898.8	7695.1
TF.IDF	53.9 (35.2%)	68041 (-28.1%)	15916.4 (-5.9%)	53.9 (35.2%)	42494.4 (18.1%)	9024.3 (-17.3%)
TF.IDF+MMR	**12.7 (84.7%)**	49746.5 (6.4%)	15634 (-4.0%)	**12.7 (84.7%)**	**9905.3 (80.9%)**	**2832.8 (63.2%)**
Okapi	**12.7 (84.7%)**	51973.8 (2.2%)	15536.1 (-3.4%)	**12.7 (84.7%)**	50407.9 (2.9%)	7483.5 (2.7%)
Okapi+MMR	548 (-558.9%)	52171.3 (1.8%)	16080.9 (-7.0%)	548 (-558.9%)	49159.6 (5.3%)	8037.3 (-4.4%)
LDA	57 (31.4%)	**46837.4 (11.8%)**	**9262.9 (38.4%)**	57 (31.4%)	46180.8 (11.0%)	6948.8 (9.7%)
LDA+MMR	**12.7 (84.7%)**	47886.4 (9.9%)	10530.2 (29.9%)	**12.7 (84.7%)**	18754.7 (63.9%)	3386.2 (56.0%)

Table 4.6: DeBBI with RTS for parallel project execution

	Without Bug Clustering			With Bug Clustering		
	Execution Time(sec)			Execution Time(sec)		
	First	**Last**	**Average**	**First**	**Last**	**Average**
Random	83.5	53122.1	15026.5	83.5	51898.8	7695.1
Random+RTS	150.9 (-80.7%)	19495 (63.3%)	8072.8 (46.3%)	150.9 (-80.7%)	17712.4 (65.9%)	4477.7 (41.8%)
TF.IDF+RTS	303.6 (-263.6%)	21391.5 (59.7%)	10575.2 (29.6%)	303.6 (-263.6%)	15991.7 (69.2%)	5299.4 (31.1%)
TF.IDF+MMR+RTS	197 (-135.9%)	18115.2 (65.9%)	10785.8 (28.2%)	197 (-135.9%)	12176.8 (76.5%)	4142.7 (46.2%)
Okapi+RTS	197 (-13.6%)	17437 (67.2%)	10656 (29.1%)	197 (-13.6%)	14250.2 (72.5%)	5019.6 (34.8%)
Okapi+MMR+RTS	889 (-964.7%)	20250.1 (61.9%)	11076.6 (26.3%)	889 (-964.7%)	16087.6 (69.0%)	5593.4 (27.3%)
LDA+RTS	**95.7 (36.6%)**	17016.2 (68.0%)	**5352.6 (64.4%)**	**95.7 (36.6%)**	11141 (78.5%)	**3014.7 (60.8%)**
LDA+MMR+RTS	197 (-135.9%)	**15180.3 (71.4%)**	9135.5 (39.2%)	197 (-135.9%)	**8757.4 (83.1%)**	3257.4 (57.7%)

4.3.5 RQ5: Can DeBBI be generalized to other popular 3rd-party libraries besides JDK?

Besides JDK, we further use other popular libraries to thoroughly evaluate the performance of our approach. For this experiment, we cloned all Maven-based Java projects that are created between August 2008 and December 2019 on GitHub with at least one star, and finally included 56,092 unique projects that can successfully pass the build and test phases in our dataset. In total, there are 40,191 3rd-party libraries used by the projects in our client project dataset. We then sort all libraries by use frequency and randomly choose 100 libraries from the top 300 to detect BBI bugs through DeBBI.

During our manual inspection, we found there are three types of false positives reported by DeBBI: (1) failures triggered by Maven POM file specifications (e.g., the specific updated library versions are prohibited by `POM.xml`), (2) failures triggered by intended changes (e.g., due to deprecated methods/implementations), and (3) failures triggered by dependency conflicts (e.g., the library updates are not compatible with specific versions of other libraries). Types (1) and (2) have their corresponding specific stack traces with fixed patterns. Thus, we were able to develop a rule-based method in DeBBI to automatically filter them out. However, we cannot avoid the false positives from Type (3). After manually removing 22 Type (3) false positives, DeBBI reported 97 unique BBI bugs. To date, 19 bugs have been confirmed as previously unknown bugs. 54 bugs have been confirmed as previously known bugs (e.g., for COLLECTIONS-721 [24]), while all the other bug reports are still under active discussion. Interestingly, among the bug reports still under discussion, some reports have already been confirmed by other users (e.g. *"Experiencing same issue."* for reflection-277 [31]) even though not yet confirmed by the actual library developers.

Quantitative analysis. Due to the space limitation, we only present partial experimental results for the library projects with confirmed previously unknown BBI bugs in Table 4.7. In the table, Columns 1-4 list all the libraries, the number of corresponding GitHub Stars, the number of client projects from our dataset using the corresponding libraries, and the revision ranges that we use to detect BBIs. Columns 5-7 further present the number of unique unknown, known, and under discussion BBI bugs reported by DeBBI for this subset of libraries. Columns 8-13 present the **First**, **Last**, and **Average** values in terms of the number of test executions and execution time for our default LDA+MMR+RTS technique (with improvement over the Random technique shown in the parenthesis). The experiment parameters used are the same as our JDK experiment. From the table, we can observe that DeBBI can consistently improve the BBI detection efficiency in all traced metrics, further demonstrating the effectiveness of DeBBI.

Qualitative analysis. For the 19 confirmed previously unknown BBI bugs, developers quickly fixed the buggy code for 4 of them, and even added our reported test case in their regression test suites for 3 of them. For example, Figure 4.7 shows the test for issue Assertj-core-1751 [23]. Method `containsOnlyKeys` cannot handle the case when the `containsOnlyKeys` API is invoked on a `Map` with key type `Path`. This test is challenging to generate automatically due to the special corner case, while DeBBI is able to directly obtain such tests for free from client projects, demonstrating the promising future of DeBBI. Interestingly, at first one developer found it too difficult to fix it and wanted to just add a breaking-change notice; later on, another developer proposed a solution to finally fix it. Issue Commons-vfs-739 [32] is triggered when using Apache Commons-vfs to parse a MapR File System file path (shown in Figure 4.8). It is also challenging to generate this test automatically since the bug will be triggered only when the first two parameters for method `parseUri` are both `null` and `URI` includes the substring ":///". Furthermore, issue Jsoup-1274 [28] from library `Jsoup`, a widely used Java HTML parser, is incurred by the change of the method `select` – the developers forgot to deal with the situation when the end of the string in method `select` is a space (shown in Figure 4.9). The method `select` should trim the space first and continue to parse the string, but it throws an exception. DeBBI is able to detect it through a special test case that used `Jsoup` to parse a specific string followed by a whitespace. The developers were also quite active in fixing issue mybatis-spring-427 [30] reported by DeBBI, saying: *"Thanks for your report! This issue is bug(This issue was included by 5ca5f2d). We will revert it at 2.0.4."*

```
@Test
public void demo() {
    Map<Path, String> test = new HashMap<Path, String>();
    Path path = Paths.get("/tmp/test/file");
    test.put(path, "pathMD5");
    assertThat(test)
    .containsOnlyKeys(path)
    .containsValue("pathMD5");
}
```

Figure 4.7: Assertj-core-1751 [23] triggering test

```
@Test
public void demo() throws FileSystemException{
    final String URI = "maprfs:///";
    UrlFileNameParser parser = new UrlFileNameParser();
    FileName name = parser.parseUri(null, null, URI);
    assertEquals(URI, name.getURI());
}
```

Figure 4.8: Commons-vfs-739 [32] triggering test

```
@Test
public void demo(){
    String content = "<p> Select Test";
    StringBuilder bodyHtml = new StringBuilder();
    bodyHtml.append(content);
    Document document = Jsoup.parse(bodyHtml.toString());
    StringBuilder nav = new StringBuilder();
    Elements bodyElements = document.select("body > * ");
}
```

Figure 4.9: Jsoup-1274 [28] triggering test

11 other confirmed BBI bugs are mitigated by the developers via changing the documents, since the developers did not realize they were BBI bugs until we submitted the reports and could not undo the change or fix the code. These BBI bugs were mitigated by adding an announcement in the corresponding documents. For example, the following comment is from the issue java-jwt-376 [26]:

"You are correct that this would be a breaking change, so should have been targeted at a future major version or at the very least called out the breaking change in the CHANGELOG.md

Table 4.7: Effectiveness of DeBBI for Other Libraries

Project	Stars	Client Num	Revision	Bug Num			Execution Time(min)			Test Case		
				Unkown	Known	Discussion	First	Last	Average	First	Last	Average
Commons-io	616	4,308	2.1 - 2.6	1	0	3	5.29 (99.62%)	408.45 (93.50%)	181.6 (95.37%)	245 (99.39%)	3525 (97.49%)	1428 (98.44%)
assertj-core	1,689	1,129	3.8.0 - 3.14.0	2	8	0	0.04 (99.99%)	211.6 (89.01%)	130.86 (88.60%)	1 (99.99%)	5252 (96.01%)	3767 (93.66%)
lombok	8,832	2,721	1.16.14 - 1.18.10	1	10	0	0.34 (99.31%)	227.74 (92.20%)	72.21 (95.39%)	1 (99.79%)	421 (99.25%)	175 (99.41%)
commons-vfs	103	39	2.2 - 2.6.0	1	0	0	0.96 (98.21%)	0.96 (98.21%)	0.96 (98.21%)	112 (92.21%)	112 (92.21%)	112 (92.21%)
jsoup	7,650	575	1.9.2 - 1.12.1	1	2	0	1.64 (99.60%)	39.49 (93.96%)	17.8 (96.56%)	81 (98.18%)	399 (96.25%)	243 (96.46%)
mybatis-spring	1,992	987	1.3.2 - 2.0.3	1	2	0	1.67 (97.30%)	48.29 (91.64%)	32.68 (90.23%)	1 (99.12%)	9 (99.92%)	6 (99.92%)
HttpAsyncClient	904	125	4.1.3 - 4.1.4	1	1	0	4.14 (97.27%)	7.9 (94.91%)	6.02 (96.08%)	55 (98.66%)	78 (98.14%)	66 (98.41%)
JENA	618	59	3.12.0 - 3.14.0	1	0	0	1.27 (98.52%)	1.27 (98.52%)	1.27 (98.52%)	13 (98.57%)	13 (98.57%)	13 (98.57%)
ognl	111	70	3.1 - 3.2.12	1	2	0	0.8 (97.28%)	3.52 (95.66%)	2.34 (95.81%)	5 (99.03%)	37 (98.20%)	19 (98.26%)
asciidoctorj	445	27	1.5.3 - 2.2.0	2	0	0	3.92 (92.09%)	4.4 (92.86%)	4.16 (92.52%)	6 (98.18%)	6 (98.27%)	6 (98.22%)
mybatis	12,730	1,135	3.1.1 - 3.5.3	1	7	0	1.18 (97.85%)	54.67 (93.57%)	11.88 (96.30%)	13 (92.61%)	106 (99.26%)	68 (98.71%)
java-jwt	3,323	119	3.2.0 - 3.6.0	1	1	0	0.63 (85.80%)	4.47 (96.87%)	2.55 (96.53%)	7 (77.42%)	21 (98.47%)	14 (98.00%)
mybatis-generator	4,105	202	1.3.5 - 1.4.0	1	1	1	0.27 (95.15%)	0.66 (98.95%)	0.51 (97.95%)	3 (62.50%)	3 (97.35%)	3 (93.18%)
jOOQ	3,646	88	3.9.0 - 3.12.4	1	1	1	0.74 (97.49%)	6.14 (97.22%)	3.75 (97.11%)	3 (99.23%)	27 (98.77%)	16 (98.75%)
bcpkix-jdk15on	1,110	122	1.5.9 - 1.6.4	1	0	0	12.3 (94.66%)	12.3 (94.66%)	12.3 (94.66%)	169 (97.67%)	169 (97.67%)	169 (97.67%)
activiti-engine	6,239	39	6.0.0 - 7.1.0	1	0	0	0.67 (96.20%)	0.67 (96.20%)	0.67 (96.20%)	1 (98.15%)	1 (98.15%)	1 (98.15%)
extentreports	517	43	3.0.7 - 4.1.2	1	0	0	3.22 (70.51%)	3.22 (70.51%)	3.22 (70.51%)	36 (86.86%)	36 (86.86%)	36 (86.86%)

file. Unfortunately, at this point we cannot undo the change without breaking others who are not handling the UnsupportedEncodingException. We should update the Change log, so keeping this issue open to address that. Apologies for the inconvenience, and thank you for raising this."

For the remaining 4 confirmed BBI bugs, issues lombok-2320 [29] and HttpAsyncClient-159 [25] cannot be easily fixed by the developers for the moment. For example, the Apache HttpAsyncClient developers said:

"There is no much we can do about it now. If we remove the offending constructor to restore full compatibility with 4.1.3 we will break full compatibility with 4.1.4."

The other 2 unfixed bugs are from Apache Commons-io and Apache Jena. They confirmed our reported BBI bugs are source incompatibility, but cannot afford to fix them. For example, the Apache Jena [27] developers said:

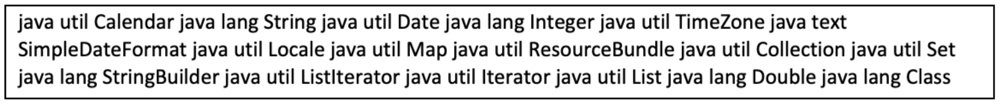
"We try to migrate gracefully, and it is a compile time error. There is a balance between compatibility and building up technical debt. Change away from use of FastDateFormat was forced on the code (staying at the old version forever is not an option). Sometimes, our understanding of what users do, and do not use, is incomplete. "

4.4 Discussions

Availability of Client Software In our experiment, due to the prevalent usage of JDK, we were able to collect 2,953 client software projects, and ran unit testing on them over JDK 8 and 9 to detect failures. One doubt on the applicability of our approach is whether there are also many client software projects for other libraries so that prioritization is necessary. Our observation is that the popular frameworks that require extensive incompatibility detection typically have lots of client software project available. For example, Android SDK, Apache software, Eclipse API, and Chrome API all have thousands of client projects in GitHub (as confirmed in RQ5). On the other hand, due to the popularity of modern build systems (Gradle/Maven) and the corresponding central repositories, even ordinary projects can have a large number of client projects on the central repositories. Such modern build systems support fully automated client project retrieval, build, and test. Thus, we can easily apply DeBBI in a fully automated way[2].

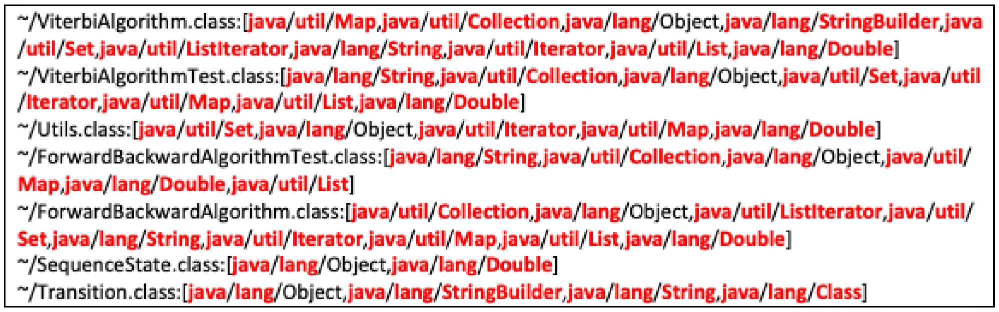
Effectiveness of Client Software Testing Another issue with client software testing is whether it is helpful when a large regression test suite is already available. From our experiment, we can see that 79 JDK incompatibility bugs can be detected if client software testing is applied before Java 9.0.0 is released. These bugs are confirmed by JDK developers in 9.0.1, and cannot be detected by the large regression test suite of JDK. Another benefit of client software testing is that it always finds real bugs. Although regression testing may also detect incompatibilities, the ones detected may be on a cold spot of API that is never used by real client software, or triggered by a method-invocation sequence that is never used by client software developers. In contrast, the incompatibilities detected by client software testing usually indicate important bugs of the library or the client software.

Why does DeBBI work? A naive approach for ranking client projects would be simply



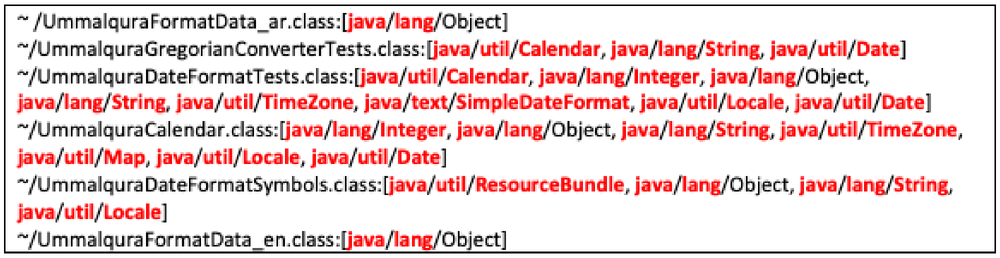
```
java util Calendar java lang String java util Date java lang Integer java util TimeZone java text
SimpleDateFormat java util Locale java util Map java util ResourceBundle java util Collection java util Set
java lang StringBuilder java util ListIterator java util Iterator java util List java lang Double java lang Class
```

Figure 4.10: Example changed JDK query



```
~/ViterbiAlgorithm.class:[java/util/Map,java/util/Collection,java/lang/Object,java/lang/StringBuilder,java
/util/Set,java/util/ListIterator,java/lang/String,java/util/Iterator,java/util/List,java/lang/Double]
~/ViterbiAlgorithmTest.class:[java/lang/String,java/util/Collection,java/lang/Object,java/util/Set,java/util
/Iterator,java/util/Map,java/util/List,java/lang/Double]
~/Utils.class:[java/util/Set,java/lang/Object,java/util/Iterator,java/util/Map,java/lang/Double]
~/ForwardBackwardAlgorithmTest.class:[java/lang/String,java/util/Collection,java/lang/Object,java/util/
Map,java/lang/Double,java/util/List]
~/ForwardBackwardAlgorithm.class:[java/util/Collection,java/lang/Object,java/util/ListIterator,java/util/
Set,java/lang/String,java/util/Iterator,java/util/Map,java/util/List,java/lang/Double]
~/SequenceState.class:[java/lang/Object,java/lang/Double]
~/Transition.class:[java/lang/Object,java/lang/StringBuilder,java/lang/String,java/lang/Class]
```

Figure 4.11: Project hmm-lib JDK usage



```
~ /UmmalquraFormatData_ar.class:[java/lang/Object]
~/UmmalquraGregorianConverterTests.class:[java/util/Calendar, java/lang/String, java/util/Date]
~/UmmalquraDateFormatTests.class:[java/util/Calendar, java/lang/Integer, java/lang/Object,
java/lang/String, java/util/TimeZone, java/text/SimpleDateFormat, java/util/Locale, java/util/Date]
~/UmmalquraCalendar.class:[java/lang/Integer, java/lang/Object, java/lang/String, java/util/TimeZone,
java/util/Map, java/util/Locale, java/util/Date]
~/UmmalquraDateFormatSymbols.class:[java/util/ResourceBundle, java/lang/Object, java/lang/String,
java/util/Locale]
~/UmmalquraFormatData_en.class:[java/lang/Object]
```

Figure 4.12: Project ummalqura-calendar JDK usage

counting the number of API terms used by each client project. In contrast to simply counting API term frequency, our DeBBI adopts information retrieval, which not only counts API term frequency, but also considers API importance, diversity, and textual information. For example, there are two JDK client projects hmm-lib [3] and ummalqura-calendar [15] from our data set. Figure 4.10 shows the portion of changed JDK query which is related to these two client projects, while Figures 4.11 and 4.12 show the JDK usage of the client projects. Interestingly, we can see many terms (highlighted in bold) matching terms in query. If we only count the term frequency, hmm-lib with 125 term matches should have a higher priority than ummalqura-calendar that only has 67 term matches. However, in our DeBBI(TF.IDF),

[2]We can also afford discarding failing client projects as online repositories provide a huge candidate project set.

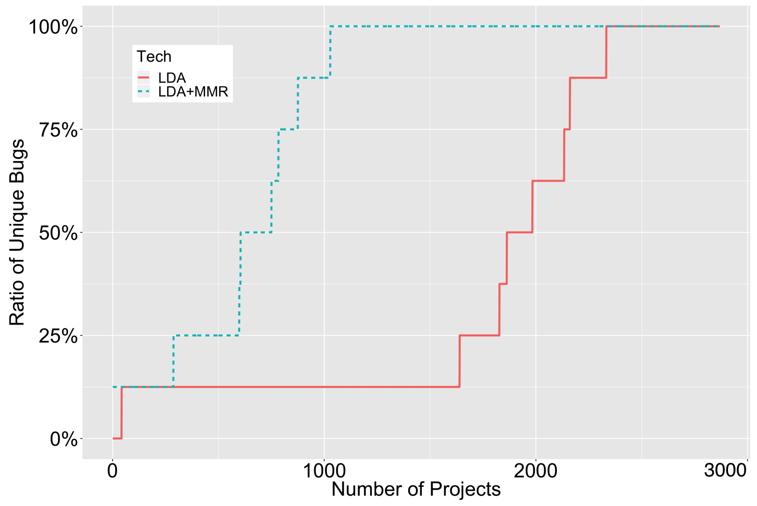


Figure 4.13: Accumulated bugs detected

hmm-lib is ranked at 2,760 with no bug and ummalqura-calendar is ranked at 442 with a real BBI issue (BugID: JDK-8008577[3], triggered by the different English locale date-time long formats between JDK 8 and JDK 9). The reason is that TF.IDF also considers the importance of low-frequency terms Locale and Date.

Why do we need diversity enhancement?

For a given query, an information retrieval system can give us a ranked list of documents all of which are relevant to the query. However, they might be all the same or very similar. This is a classic diversity or novelty issue in information retrieval. In our scenario, if DeBBI uses only traditional information retrieval technique, the top-ranked client projects might detect the same bugs repeatedly. Therefore, we use the MMR algorithm to solve this issue to detect more unique bugs faster. In Figure 4.13, the solid and dashed lines present the effectiveness of detecting unique BBI bugs for JDK when applying LDA and LDA+MMR, respectively. The *x*-axis is the number of projects we need to run, the *y*-axis is the percentage of unique JDK BBI bugs we can detect. We observed that LDA+MMR found the first unique bug

[3]`https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8008577`

at the 1st position and the last unique bug at the 1029rd position, while LDA found the first/last unique bug at the 43rd/2333rd position, demonstrating the effectiveness of diversity enhancement for further boosting DeBBI.

4.5 Threats to Validity

The major internal threat to our evaluation is whether our ground truth on incompatibility bugs is correct. For JDK, although large-scale client testing reveals a lot of test failures, their causes are different and may not always indicate incompatibilities of JDK. For example, Raemaekers et al. [141] observed that library-breaking changes have a huge impact on project compilation. To reduce this threat, we use the test failures that are fixed when using Java 9.0.1 as the ground truth because they are incompatibility issues confirmed by JDK developers. This solution is not perfect as we may miss some real JDK incompatibilities and bugs that are not noticed and confirmed by JDK developers. For the popular 3rd-party libraries, we manually inspected all the reported cases (since they are more affordable than the JDK experiments) to confirm the ground truth, and also filed corresponding bug reports for the software developers to confirm. The major external threat to our evaluation is whether our approach may be generalized to libraries other than the studied ones. It should be noted that JDK is not a single library but a collection of tens of Java packages and even libraries developed by the 3rd-party such as SAXP libraries by XML-DEV and DOM libraries by W3C. To reduce such threats, we have also applied DeBBI to detect BBIs for other widely used 3rd-party libraries from GitHub. In the future, we further plan to further apply our DeBBI to other widely-used libraries such as the Android SDK.

CHAPTER 5

FAST AND PRECISE ON-THE-FLY PATCH VALIDATION FOR ALL

Besides exploring speeding up software testing in Chapter 3 and Chapter 4, we also applied our faster revision testing in software debugging. We aim to decrease the patch validation time to speed up APR approaches.

Software bugs are inevitable in modern software systems, costing trillions of dollars in financial loss and affecting billions of people [45]. Meanwhile, software debugging can be extremely challenging and costly, consuming over half of the software development time and resources [162]. Therefore, a large body of research efforts have been dedicated to automated debugging techniques [68, 120, 176]. Among the existing debugging techniques, Automated Program Repair [73] (APR) techniques hold the promise of reducing debugging effort by suggesting likely patches for buggy programs with minimal human intervention, and have been extensively studied in the recent decade. Please refer to the recent surveys on APR for more details [68, 120].

Generate-and-validate (G&V) APR refers to a practical category of APR techniques that attempt to fix the bugs by first generating a pool of patches and then validating the patches via certain rules and/or checks [68]. A patch is said to be *plausible* if it passes all the checks. Ideally, we would apply formal verification [127] techniques to guarantee correctness of generated patches. However, in practice, formal specifications are often unavailable for real-world projects, thus making formal verification infeasible. In contrast, testing is the prevalent, economic methodology of getting more confidence about the quality of software [37]. Therefore, the vast majority of recent G&V APR techniques leverage developer tests as the criteria for checking correctness of the generated patches [68], i.e., *test-based* G&V APR.

Two main costs are associated with such test-based G&V APR techniques: (1) the cost of manipulating program code to fabricate/generate patches based on certain transformation rules; (2) repeated executions of all the developer tests to identify plausible patches for the bugs under fixing. Since the search space for APR is infinite and it is impossible to triage the elements of this search space due to theoretical limits, test-based G&V APR techniques usually lack clear guidance and often act in a rather brute-force fashion: they usually generate a huge pool of patches to be validated and the larger the program the larger the set of patches to be generated and validated. This suggests that the speed of patch generation and validation plays a key role in scalability of the APR techniques, which is one of the most important challenges in designing practical APR techniques [57]. Therefore, apart from introducing new and/or more effective transformation rules, some APR techniques have been proposed to mitigate the aforementioned costs. For example, JAID [51] uses mutation schema to fabricate meta-programs that bundle multiple patches in a single source file, while SketchFix [81] uses sketches [100] to achieve a similar effect. However, such techniques mainly aim to speed up the patch generation time, while patch validation time has been shown to be dominant during APR [118]. Most recently, PraPR [69] aims to reduce both patch generation and validation time by modifying program code directly at the bytecode level with *on-the-fly patch validation*, which directly allows multiple bytecode-level patches to be tested within the same JVM process. However, bytecode-level APR is not flexible (e.g., large-scope changes can be extremely hard to implement at the bytecode level) and fails to fix many bugs that can be fixed at the source-code level [69]; furthermore, PraPR requires decompilation (which may be imprecise or even fail) to decompile the bytecode-level patches for manual inspection. In fact, all other popular general-purpose G&V APR techniques fix at the source code level.

In this paper, we propose a unified test-based patch validation framework, named UniAPR, to empirically study the impact of *on-the-fly* patch validation for state-of-the-art source-code-level APR techniques. While existing source-code-level APR usually restarts a new JVM process for each patch, our on-the-fly patch validation aims to use a single JVM process for patch validation, as much as possible, and leverages JVM's dynamic class redefinition feature (a.k.a. the HotSwap mechanism and Java Agent technology [55]) to only reload the patched bytecode classes on-the-fly for each patch. In this way, UniAPR not only avoids reloading (also including linking and initializing) all used classes for each patch (i.e., only reloading the *patched* bytecode files), but also can avoid the unnecessary JVM warm-up time (e.g., the accumulated JVM profiling information across patches enables more and more code to be JIT-optimized and the already JIT-optimized code can also be shared across patches).

UniAPR has been implemented as a fully automated Maven [33] plugin (available at [165]), to which almost all existing state-of-the-art Java APR tools can be attached in the form of patch generation *add-ons*. We have constructed add-ons for representative APR tools from different APR families. Specifically, we have constructed add-ons for CapGen [175], SimFix [85], and ACS [179] that are modern representatives of template-/pattern-based [**? ?**], heuristic-based [41, 95], and constraint-based [126, 180] techniques. Our empirical study shows for the first time that on-the-fly patch validation can often speed up state-of-the-art APR systems by over an order of magnitude, enabling all existing APR techniques to explore a larger search space to fix more bugs in the near future.

Furthermore, our study (Section 5.3.1) shows the first empirical evidence that when sharing JVM across multiple patches, the global JVM state may be *polluted* by earlier patch executions, making later patch execution results unreliable. For example, some patches may modify some static fields, which are used by some later patches sharing the same JVM. Therefore, we further propose the first solution to address such imprecision problem by isolating patch executions via resetting JVM states after each patch execution using runtime bytecode transformation. Our experimental results show that our UniAPR with JVM reset is able to the avoid imprecision/unsoundness of vanilla on-the-fly patch validation with negligible overhead.

We envision a future wherein all existing APR tools (like SimFix [85], CapGen [175], and ACS [179]) and major APR frameworks (like ASTOR [115] and Repairnator [122]) are leveraging this framework for patch validation. In this way, researchers will only need to focus on devising more effective algorithms for better exploring the patch search space, rather than spending time on developing their own components for patch validation, as we can have a unified, generic, and much faster framework for all. In summary, this paper makes the following contributions:

- **Framework.** We introduce the first unified on-the-fly patch validation framework, UniAPR, to empirically study the impact of on-the-fly patch validation for state-of-the-art source-code-level APR techniques.

- **Technique.** We show the first empirical evidence that on-the-fly patch validation can be imprecise/unsound, and introduce a new technique to reset the JVM state right after each patch execution to address such issue.

- **Implementation.** We have implemented on-the-fly patch validation based on the JVM HotSwap mechanism and Java Agent technology [55], and implemented the JVM-reset technique based on the ASM bytecode manipulation framework [128]; the overall UniAPR tool has been implemented as a practical Maven plugin [165], and can accept different APR techniques as patch generation add-ons.

- **Empirical Study.** We conduct a large-scale study of the effectiveness of UniAPR on its interaction with state-of-the-art APR systems from three different APR families, demonstrating that UniAPR can often speed up state-of-the-art APR by over an order of magnitude (without validation imprecision/unsoundness). Furthermore, the study results also indicate that UniAPR can serve as a unified platform to naturally support hybrid APR to directly combine the strengths of different APR tools.

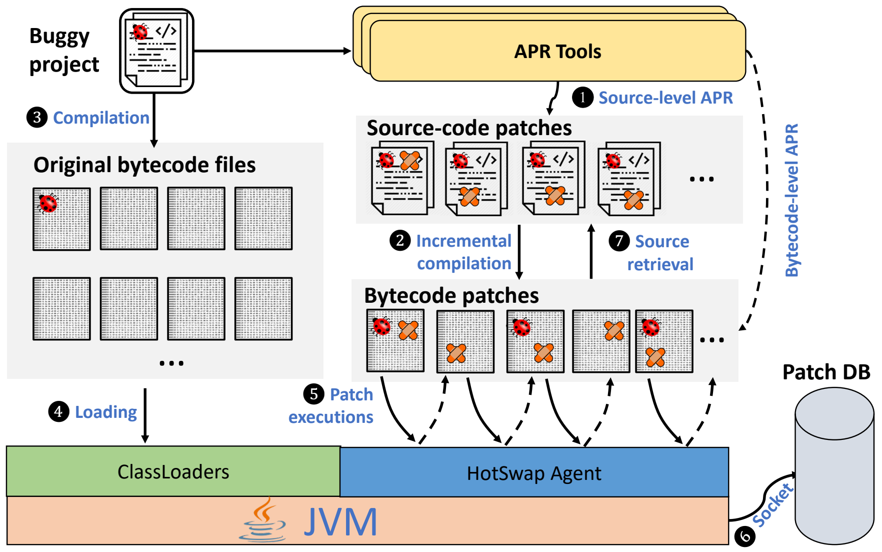


Figure 5.1: UniAPR workflow

5.1 Approach

5.1.1 Overview

Figure 5.1 depicts an the overall flow of our UniAPR framework. According to the figure, given a buggy project, UniAPR first leverages any of the existing APR tools (integrated as UniAPR add-ons) to generate source-code level patches (marked with ❶). Then, UniAPR performs incremental compilation to compile the patched source file(s) by each patch into bytecode file(s) (marked with ❷). Note that, UniAPR is a unified framework and can also directly take the bytecode patches generated by the PraPR [69] (and future) bytecode APR technique (marked with the dashed line directly connecting APR tools into bytecode patches). In this way, UniAPR has a pool of bytecode patches for patch validation. Also note that besides constructed before patch validation, the patch pool can also be continuously generated during the patch-validation process[1]; in either way, UniAPR's reduction on patch-validation time is also not affected.

[1]If patches are continuously generated, the patch-validation component needs to obtain the live stream of patch information from the running patch-generation component (e.g., via lightweight socket connections).

During the actual patch validation, UniAPR first compiles the entire buggy project into bytecode files (i.e., `.class` files), and then loads all the bytecode files into the JVM through JVM class loaders (marked with ❸ and ❹ in the figure). Note that these two steps are exactly the same as executing the original tests for the buggy project. Since all the bytecode files for the original project are loaded within the JVM, when validating each patch, UniAPR only reloads the patched bytecode file(s) by that particular patch via the Java Agent technology and HotSpot mechanism, marked with ❺ (as the other unpatched bytecode files are already within the JVM). Then, the test driver can be triggered to execute the tests to validate against the patch without restarting a new JVM. After all tests are done for this patch execution, UniAPR will replace the patched bytecode file(s) with the original one(s) to revert to the original version. Furthermore, UniAPR also resets the global JVM states to prepare a clean JVM environment for the next patch execution (marked with the short dashed lines). The same process is repeated for each patch. Finally, the patch validation results will be stored into the patch execution database via socket connections (marked with ❻). Note that for any plausible patch that can pass all the tests, UniAPR will directly retrieve the original source-level patch for manual inspection (marked with ❼) in case the patch was generated by source-level APR.

We have already constructed add-ons for three different APR tools representing three different families of APR techniques. These add-ons include CapGen [175] (representing pattern/template-based APR techniques), SimFix [85] (representing heuristic-based techniques), and ACS [179] (representing constraint-based techniques). Of course, users of UniAPR can also easily build new patch generation add-ons for other APR tools. For existing APR tools, this can be easily done by modifying their source code so that the tools abandon validation of patches after generating/compiling them.

Next, we will talk about our detailed design for *fast* patch validation via on-the-fly patching (Section 5.1.2) as well as *precise* patch validation via JVM reset (Section 5.1.3).

5.1.2 Fast Patch Validation via On-the-fly Patching

Algorithm 2 is a simplified description of the steps that *vanilla* UniAPR (without JVM-reset) takes in order to validate candidate patches on-the-fly. The algorithm takes as inputs the original buggy program $\mathcal{P}$, its test suite $\mathcal{T}$, and the set of candidate patches $\mathbb{P}$ generated by any APR technique[2]. The output is a map, $\mathcal{R}$, that maps each patch into its corresponding execution result. The overall UniAPR algorithm is rather simple. UniAPR first initializes all patch execution results as unknown (Line 2). Then, UniAPR gets into the loop body and obtains the set of patches still with unknown execution results (Line 4). If there is no such patches, the algorithm simply returns since all the patches have been validated. Otherwise, it means this is the first iteration or the earlier JVM process gets terminated abnormally (e.g., due to timeout or JVM crash). In either case, UniAPR will create a new JVM process (Line 7) to evaluate the remaining patches (Line 8).

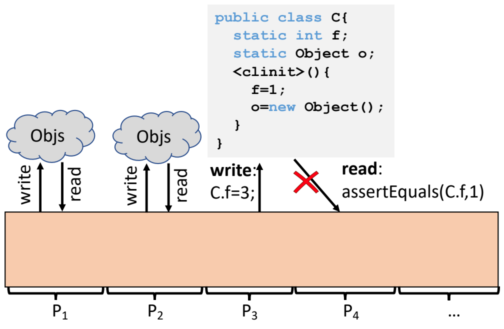
We next talk about the detailed `validate` function, which takes the remaining patches, the original test suite, and a new JVM as input. For each remaining patch $\mathcal{P}'$, the function first obtains the patched class name(s) $\mathcal{C}_{patched}$ and patched bytecode file(s) $\mathcal{F}_{patched}$ within $\mathcal{P}'$ (Lines 11 and 12). Then, the function leverages our HotSwap Agent to replace the bytecode file(s) under the same class name(s) as $\mathcal{C}_{patched}$ with the patched bytecode file(s) $\mathcal{F}_{patched}$; it also stores the replaced bytecode file(s) as $\mathcal{F}_{orig}$ to recover it later (Line 13). Note that our implementation will explicitly load the corresponding class(es) to patch (e.g., via `Class.forName()`) if they are not yet available before swapping. In this way, the function can now execute the tests within this JVM to validate the current patch since the patched bytecode file(s) has already been loaded (Lines 14-26). If the execution for a test finishes normally, its status will be marked as `Plausible` or `Non-Plausible` (Lines 16-19); otherwise,

[2]Note that here we assume that $\mathbb{P}$ is available before patch validation for the ease of presentation, but our overall approach is general and can also easily handle the case where $\mathbb{P}$ is continuously constructed during patch validation.

Algorithm 2: Vanilla on-the-fly patch validation

```
Input: Original buggy program $\mathcal{P}$, test suite $\mathcal{T}$, and set of candidate patches $\mathbb{P}$
Output: Validation status $\mathcal{R} : \mathbb{P} \rightarrow \{\texttt{PLAUSIBLE}, \texttt{NON}-\texttt{PLAUSIBLE}, \texttt{ERROR}\}$
1  begin
2      $\mathcal{R} \leftarrow \mathbb{P} \times \{\texttt{UNKNOWN}\}$ ; // initialize result function
3      while True do
4          $\mathbb{P}_{left} \leftarrow \{\mathcal{P}' \mid \mathcal{P}' \in \mathbb{P} \wedge \mathcal{R}(\mathcal{P}') = \texttt{UNKNOWN}\}$// get all the left patches not yet validated
5          if $\mathbb{P}_{left} = \emptyset$ then
6              return $\mathcal{R}$ // return if no left patches
7          $\mathcal{JVM} \leftarrow$ createJVMProcess()// create a new JVM
8          validate($\mathbb{P}_{left}, \mathcal{T}, \mathcal{JVM}$)) // validate the left patches on the new JVM
9  function validate($\mathbb{P}_{left}, \mathcal{T}, \mathcal{JVM}$):
10     for $\mathcal{P}'$ in $\mathbb{P}_{left}$ do
11         $\mathcal{C}_{patched} \leftarrow$ patchedClassNames($\mathcal{P}'$)
12         $\mathcal{F}_{patched} \leftarrow$ patchedBytecodeFiles($\mathcal{P}'$) // Swap in the patched bytecode files
13         $\mathcal{F}_{orig} \leftarrow$ HotSwapAgent.swap($\mathcal{JVM}, \mathcal{C}_{\text{patched}}, \mathcal{F}_{patched}$)
14         for $t$ in $\mathcal{T}$ do
15             try:
16                 if run($\mathcal{JVM}, t$) = FAILING then
17                     status $\leftarrow$ NON − PLAUSIBLE
18                 else
19                     status $\leftarrow$ PLAUSIBLE
20             catch TimeOutException, MemoryError:
21                 status $\leftarrow$ ERROR
22             $\mathcal{R} \leftarrow \mathcal{R} \cup \{\mathcal{P}' \rightarrow status\}$
23             if status = NON-PLAUSIBLE then
24                 break // continue with the next patch when current one is falsified
25             if status = ERROR then
26                 return // restart a new JVM when this current one timed out or crashed
           // Swap back the original bytecode files
27         HotSwapAgent.swap($\mathcal{JVM}, \mathcal{C}_{patched}, \mathcal{F}_{orig}$)
```

the status will be marked as `Error`, e.g., due to timeout or JVM crash (Lines 20-21). Then, $\mathcal{P}'$'s status will be updated in $\mathcal{R}$ (Line 22). If the current status is `Non-Plausible`, the function will abort the remaining test executions for the current patch since it has been falsified, and move on to the next patch (Line 24); if the current status is `Error`, the function will return to the main algorithm (Line 26), which will restart the JVM. When the validation for the current patch finishes without the `Error` status, the function will also recover the patched bytecode file(s) into the original one(s) to facilitate the next patch validation (Line 27).



```
public class C{
  static int f;
  static Object o;
  <clinit>(){
    f=1;
    o=new Object();
  }
}
```

Figure 5.2: Imprecision under vanilla UniAPR

5.1.3 Precise Patch Validation via JVM Reset

Limitations for vanilla on-the-fly patch validation

The vanilla on-the-fly patch validation presented in Section 5.1.2 works for most patches of most buggy projects. The basic process can be illustrated via Figure 5.2. In the figure, each patch (e.g., from $\mathcal{P}_1$ to $\mathcal{P}_4$) gets executed sequentially on the same JVM. It would be okay if every patch accesses and modifies the objects created by itself, e.g., $\mathcal{P}_1$ and $\mathcal{P}_2$ will not affect each other and the vanilla on-the-fly patch validation results for $\mathcal{P}_1$ and $\mathcal{P}_2$ will be the same as the ground-truth patch validation results. However, it will be problematic if one patch writes to some global space (e.g., static fields) and later on some other patch(es) reads from that global space. In this way, earlier patch executions will affect later patch executions, and we call such global space *pollution sites*. To illustrate, in Figure 5.2, $\mathcal{P}_3$ write to some static field `C.f`, which is later on accessed by $\mathcal{P}_4$. Due to the existence of such pollution site, the execution results for $\mathcal{P}_4$ will no longer be precise, e.g., its assertion will now fail since `C.f` is no longer 1, although it may be a correct patch.

Technical challenges

We observe that accesses to static class fields are the main reason leading to imprecise on-the-fly patch validation. Ideally, we only need to reset the values for the static fields that

```
// org.joda.time.TestYearMonthDay_Constructors.java
public class TestYearMonthDay_Constructors extends TestCase {
    private static final DateTimeZone PARIS = DateTimeZone.forID("Europe/Paris");
    private static final DateTimeZone LONDON = DateTimeZone.forID("Europe/London");
    private static final Chronology GREGORIAN_PARIS =
            GregorianChronology.getInstance(PARIS);
    ...
```

Figure 5.3: Static field dependency

```
// org.joda.time.TestDateTime_Basics.java
public class TestDateTime_Basics extends TestCase {
    private static final ISOChronology ISO_UTC = ISOChronology.getInstanceUTC();
    ...
// org.joda.time.chrono.ISOChronology.java
public final class ISOChronology extends AssembledChronology {
    private static final ISOChronology[] cFastCache;
    static {
        cFastCache = new ISOChronology[FAST_CACHE_SIZE];
        INSTANCE_UTC = new ISOChronology(GregorianChronology.getInstanceUTC());
        cCache.put(DateTimeZone.UTC, INSTANCE_UTC);
    }
    ...
```

Figure 5.4: Static initializer dependency

may serve as pollution sites right after each patch execution. In this way, we can always have a clean JVM state to perform patch execution without restarting the JVM for each patch. However, it turns out to be rather challenging:

First, we cannot simply reset the static fields that can serve as pollution sites. The reason is that some static fields are `final` and cannot be reset directly. Furthermore, static fields may also be data-dependent on each other; thus, we have to carefully maintain their original ordering, since otherwise the program semantics may be changed. For example, shown in Figure 5.3, `final` field `GREGORIAN_PARIS` is data-dependent on another `final` field, `PARIS`, under the same class within project Joda-Time [34] from the widely studied Defects4J dataset [86]. The easiest way to keep such ordering and reset `final` fields is to simply re-invoke the original class initializer. However, according to the JVM specification, only JVM can invoke such static class initializers.

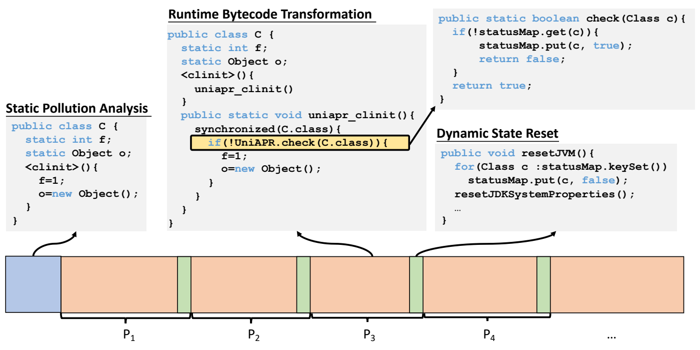
Second, simply invoking the class initializers for all classes with pollution sites may not work. For example, a naive way to reset the pollution sites is to simply trace the classes with pollution sites executed during each patch execution; then, we can simply force JVM to invoke all their class initializers after each patch execution. However, it can bring side effects because the class initializers may also depend on each other. For example, shown in Figure 5.4, within Joda-Time, the static initializer of class `TestDateTime_Basics` depends on the static initializer of `ISOChronology`. If `TestDateTime_Basics` is reinitialized earlier than `ISOChronology`, then field `ISO_UTC` will no longer be matched with the newest `ISOChronology` state. Therefore, we have to reinitialize all such classes following their original ordering as if they had been executed on a new JVM.

Based on the above analysis, we basically have two choices to implement such system: (1) customizing the underlying JVM implementation, and (2) simulating the JVM customizations at the application level. Although it would be easier to directly customize the underlying JVM implementation, the system implementation will not be applicable for other stock JVM implementations. Therefore, we choose to simulate the JVM customizations at the application level.

JVM reset via bytecode transformation

We now present our detailed approach for resetting JVM at the the application level. Inspired by prior work on speeding up traditional regression testing [42], we perform runtime bytecode transformation to simulate JVM class initializations for patch execution isolation for the first time. The overall approach is illustrated in Figure 5.5. We next present the detailed three phases as follows.

Static Pollution Analysis. Before all the patch executions, our approach performs lightweight static analysis to identify all the pollution sites within the bytecode files of all



```
public class C {
  static int f;
  static Object o;
  <clinit>(){
    f=1;
    o=new Object();
  }
}
```

```
public class C {
  static int f;
  static Object o;
  <clinit>(){
    uniapr_clinit()
  }
  public static void uniapr_clinit(){
    synchronized(C.class){
      if(!UniAPR.check(C.class)){
        f=1;
        o=new Object();
      }
    }
  }
}
```

```
public static boolean check(Class c){
  if(!statusMap.get(c)){
      statusMap.put(c, true);
      return false;
  }
  return true;
}
```

```
public void resetJVM(){
  for(Class c :statusMap.keySet())
    statusMap.put(c, false);
  resetJDKSystemProperties();
  …
}
```

Figure 5.5: On-the-fly patch validation via JVM reset

classes for the project under repair, including all the application code and 3rd-party library code. Note that we do not have to analyze the JDK library code since JDK usually provides public APIs to reset the pollution sites within the JDK, e.g., `System.setProperties(null)` can be used to reset any prior system properties and `System.setSecurityManager(null)` can be leveraged to reset prior security manager. The analysis basically returns all classes with non-`final static` fields or `final static` fields with non-primitive types (their actual object states in the heap can be changed although their actual references cannot be changed), since the states for all such static fields can be changed across patches. Shown in Figure 5.5, the blue block denotes our static analysis, and class `C` is identified since it has static fields `f` and `o` that can be mutated.

Table 5.1: Class initialization conditions

C1	T is a class and an instance of T is created
C2	T is a class and a static method declared by T is invoked.
C3	A static field declared by T is assigned
C4	A static field declared by T is used and the field is not a constant variable
C5	T is a top level class, and an assert statement lexically nested in T is executed

Runtime Bytecode Transformation. According to Java Language Specification (JSL) [130], static class initializers get invoked when any of the five conditions shown in Table 5.1 gets satisfied. Therefore, the ideal way to reinitialize the classes with pollution sites is to simply follow the JSL design. To this end, we perform runtime bytecode transformation to add class initializations right before any instance that falls in to the five conditions shown in Table 5.1.

Note that our implementation also handles the non-conventional Reflection-based accesses to such potential pollution sites. Since JVM does not allow class initialization at the application level, following prior work on speeding up traditional regression testing [42], we rename the original class initializers (i.e., `<clinit>()`) to be invoked into another customizable name (say `uniapr_clinit()`). Meanwhile, we still keep the original `<clinit>()` initializers since JVM needs that for the initial invocation; however, now `<clinit>()` initializers do not need to have any content except an invocation to the new `uniapr_clinit()`. Note that we also remove potential `final` modifiers for pollution sites during bytecode transformation to enable reinitializations of `final` non-primitive static fields. Since this is done at the bytecode level after compilation, the original compiler will still ensure that such `final` fields cannot be changed during the actual compilation phase.

Now, we will be able to reinitialize classes via invoking the corresponding `uniapr_clinit()` methods. However, JVM only initializes the same class once within the same JVM, while now `uniapr_clinit()` will be executed for each instance satisfying the five conditions in Table 5.1. Therefore, we need to add the dynamic check to ensure that each class only gets (re)initialized once for each patch execution. Shown in Figure 5.5, the orange blocks denote different patch executions. During each patch execution, the classes with pollution sites will be transformed at runtime. For example, class `C` will be transformed into the code block connected with the $\mathcal{P}_3$ patch execution in Figure 5.5; the yellow line in the transformed code denotes the dynamic check to ensure that `C` is only initialized once for each patch. The pseudo code for the dynamic check is shown in the top-right of the figure: the check maintains a `ConcurrentHashMap` for the classes with pollution sites and their status (`true` means the corresponding class has been reinitialized). The entire initialization is also synchronized based on the corresponding `Class` object to handle concurrent accesses to class initializers; in fact, JVM also leverages a similar mechanism to avoid class reinitializations due to concurrency (despite implementing that at a different level). (Note that this simplified mechanism is just for illustration purpose; our actual implementation manipulates arrays with optimizations for faster and safe tracking/check.) In this way, when the first request for initializing class `C` arrives, all the other requests will be blocked. If the class has not been initialized, then only the current access will get the return value of `false` to reinitialize `C`, while all other other requests will get the `true` value and skip the static class initialization. Furthermore, the static class initializers get invoked following the same order as if they were invoked in a new JVM.

Dynamic State Reset. After each patch execution, our approach will reset the state for the classes within the status `ConcurrentHashMap`. In this way, during the next patch execution, all the used classes within the `ConcurrentHashMap` will be reinitialized (following the check in Figure 5.5). Note that besides the application and 3rd-party classes, the JDK classes themselves may also have pollution sites. Luckily, JDK provides such common APIs to reset such pollution sites without the actual bytecode transformation. In this way, our implementation also invokes such APIs to reset potential JDK pollution sites. Please also note that our system provides a public interface for the users to customize the reset content for different projects under repair. For example, some projects may require preparing specific external resources for each patch execution, which can be easily added to our public interface. In Figure 5.5, the green strips denote the dynamic state reset, and the example reset code after $\mathcal{P}_3$'s execution simply resets the status for all classes within the status map as `false` and also resets potential JDK pollution sites within classes.

5.2 Experimental Setup

5.2.1 Dataset

We choose the Defects4J (V1.0.0) benchmark suite [86], since it contains hundreds of real-world bugs from real-world systems, and has become the most widely studied dataset for

Table 5.2: Defects4J V1.0.0 statistics

Sub.	Name	#Bugs	#Tests	LoC
Chart	JFreeChart	26	2,205	96K
Time	Joda-Time	27	4,130	28K
Lang	Apache commons-lang	65	2,245	22K
Math	Apache commons-math	106	3,602	85K
Closure	Google Closure compiler	133	7,927	90K
Total		357	20,109	321K

program repair [51, 63, 69, 85, 175] or even software debugging in general [43, 101, 102]. Table 5.2 presents the statistics for the Defects4J dataset. Column "Sub." presents the project IDs within Defects4J, while Column "Name" presents the actual project names. Column "#Bugs" presents the number of bugs collected from real-world software development for each project, while Columns "#Tests" and "LoC" present the number of tests (i.e., JUnit test methods) and the lines of code for the `HEAD` buggy version of each project.

Being a well-developed field, APR offers us a cornucopia of choices to select from. According to a recent study [105], there are 31 APR tools targeting Java programs considering two popular sources of information to identify Java APR tools: the community-led `program-repair.org` website and the living review of APR by Monperrus [121]. 17 of those Java APR tools are found to be publicly available and applicable to the widely used Defects4J benchmark suite (without additional manually collected information, e.g., potential bug locations) as of July 2019. Note that all such tools are source-level APR, since the only bytecode-level APR tool PraPR was only available after July 2019. Table 5.3 presents all such existing Java-based APR tools, which can be categorized into three main categories according to prior work [105]: heuristic-based [85, 95, 106], constraint-based [63, 180], and template-based [104, 175] repair techniques. In this work, we aims to speed up all existing source-level APR techniques via on-the-fly patch validation. Therefore, we select one representative APR tool from each of the three categories for our evaluation to demonstrate the general applicability of our UniAPR framework. All the three considered APR tools, i.e., ACS [179], SimFix [85], and

Table 5.3: Available Java APR tools for Defects4J [105]

Tool Category	Tools
Constraint-based	**ACS**, Nopol, Cardumen, Dynamoth
Heuristic-based	**SimFix**, Arja, GenProg-A, jGenProg, jKali, jMutRepair, Kali-A, RSRepair-A
Template-based	**CapGen**, TBar, AVATAR, FixMiner, kPar

CapGen [175] are highlighted in bold font in the table. For each of the selected tools, we evaluate them on all the bugs that have been reported as fixed (with correct patches) by their original papers to evaluate: (1) UniAPR effectiveness, i.e., how much speedup UniAPR can achieve, and (2) UniAPR precision, i.e., whether the patch validation results are consistent with and without UniAPR.

5.2.2 Implementation

UniAPR has been implemented as a publicly available fully automated Maven plugin [165], on which one can easily integrate any patch generation add-ons. The current implementation involves over 10K lines of Java code. As a Maven plugin, the users simply need to add the necessary plugin information into the POM file. In this way, once the users fire command: `mvn org.uniapr:uniapr-plugin:validate`, the plugin will automatically obtain all the necessary information for patch validation. It will automatically obtain the test code, source code, and 3-rd party libraries from the underlying POM file for the actual test execution. Furthermore, it will automatically load all the patches from the default `patches-pool` directory (note that the patch directory name and patch can be configured through POM as well) created by the APR add-ons for patch validation. The current UniAPR version assumes the patch directory generated by the APR add-ons to include all available patches represented by their patched bytecode files, i.e., the patch pool is constructed before patch validation. Note that, each patch may involve more than one patched bytecode file, e.g., some APR tools (such as SimFix [85]) can fix bugs with multiple edits.

During patch validation, UniAPR forks a JVM and passes all the information about the test suites and the subject programs to the child process. The process runs tests on each patch and reports their status. We use TCP Socket Connections to communicate between processes. UniAPR repeats this process of forking and receiving report results until all the patches are executed. It is worth noting that it is very easy for UniAPR to fork two or more processes to take maximum advantage of today's powerful machines. However, for a fair comparison with existing work, we always ensure that only one JVM is running patch validation at any given time stamp.

5.2.3 Experiment Settings

For each of the studied APR tools, we perform the following experiments on all the bugs that have been reported as fixed in their original papers:

First, we execute the original APR tools to trace their original patch-validation time and detailed repair results (e.g., the number of patches executed and plausible patches produced). Note that the only exception is for CapGen: digging into the decompiled CapGen code (CapGen source code is not available), we observed that CapGen excluded some (expensive) tests for certain bugs via unsafe test selection. Such unsafe test selection is inconsistent with the original paper [175], and can be dangerous (i.e., it may fail to falsify incorrect patches). Therefore, to enable a fair and realistic study, for CapGen, we build a variant for vanilla UniAPR that simply restarts a new JVM for each patch (same as CapGen) to simulate the original CapGen performance. Note that if we had presented the performance comparison between UniAPR and the original CapGen using the same reduced tests, the UniAPR speedup can be even larger because UniAPR mainly reduces the JVM-restart overhead — similar reduction on JVM overhead would yield larger overall speedup given shorter test-execution time (as the overall patch-validation time includes JVM overhead and test-execution time). For example, the average speedup achieved by UniAPR with JVM-reset on Chart bugs is 15.7X compared with the original CapGen (on the same set of reduced tests) and 8.4X compared with our simulated CapGen.

Next, we modify the studied tools and make them conform to UniAPR add-on interfaces, i.e., dumping all the generated patches into the patch directory format required by UniAPR. Then, we launch our UniAPR to validate all the patches generated by each of the studied APR tools on all the available tests, and trace the new patch validation time and results. Note that we repeat this step for both variants of UniAPR (i.e., vanilla UniAPR and UniAPR with JVM reset) to evaluate their respective performance.

To evaluate our UniAPR variants, we include the following metrics: (1) the speedup compared with the original patch validation time, measuring the effectiveness of UniAPR, and (2) the repair results compared with the original patch validation, measuring the precision of our patch validation (i.e., checking whether UniAPR fails to fix any bugs that can be fixed via traditional patch validation). All our experimentation is done on a Dell workstation with Intel Xeon CPU E5-2697 v4@2.30GHz and 98GB RAM, running Ubuntu 16.04.4 LTS and Oracle Java 64-Bit Server version 1.7.0_80.

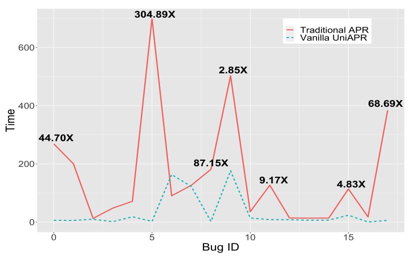
5.3 Result Analysis

To thoroughly evaluate our UniAPR framework, in this study, we aim to investigate the following two research questions.

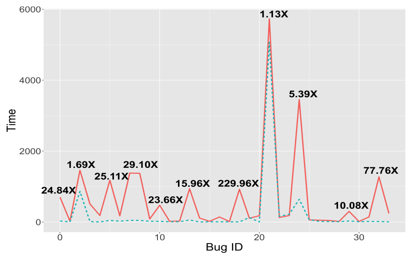
5.3.1 RQ1: How does vanilla on-the-fly patch validation perform for automated program repair?

Effectiveness

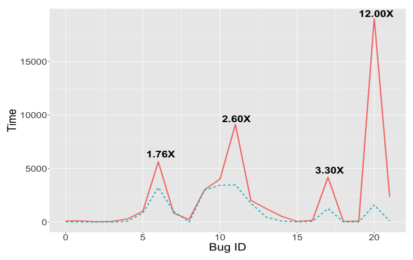
For answering this RQ, we evaluated vanilla UniAPR (i.e., without JVM-reset) that is configured to use the add-on corresponding to each studied APR tool. The main experimental



(a) ACS

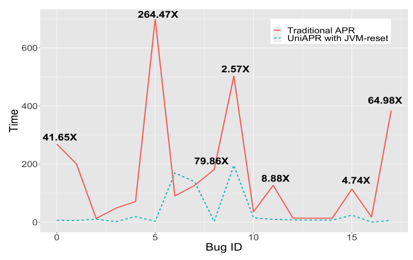


(b) SimFix

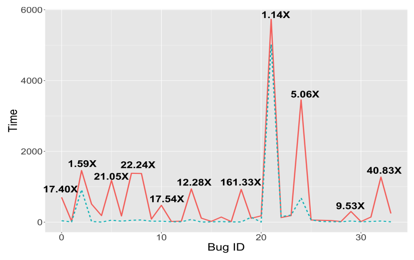


(c) CapGen

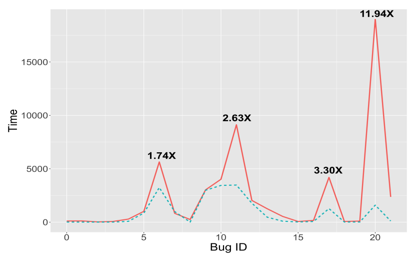
Figure 5.6: Speedup achieved by vanilla UniAPR



(a) ACS

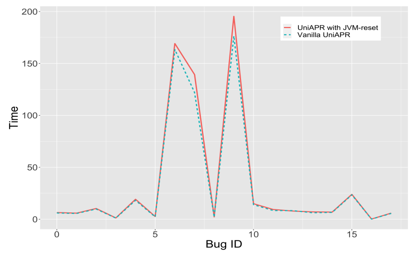


(b) SimFix

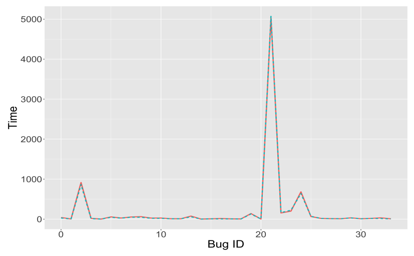


(c) CapGen

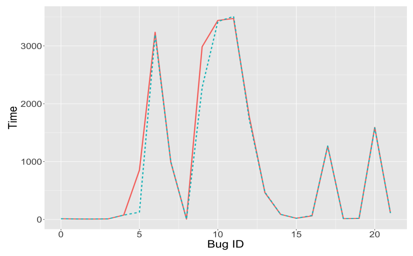
Figure 5.7: Speedup achieved by UniAPR with JVM reset



(a) ACS



(b) SimFix



(c) CapGen

Figure 5.8: JVM-reset overhead over vanilla UniAPR

results are presented in Figure 5.6. In each sub-figure, the horizontal axis presents all the bugs that have been reported to be fixed by each studied tool, while the vertical axis presents the time cost (s); the solid and dashed lines present the time cost for traditional patch validation and our vanilla UniAPR, respectively.

From the figure, we can observe that the vanilla UniAPR can substantially speed up the existing patch validation component for all state-of-the-art APR tools with almost no slowdowns. For example, when running ACS on Math-25, the traditional patch validation

Table 5.4: Inconsistent fixing results

Tool	# All	# Mismatch	Ratio (%)
CapGen	22	3	13.64%
SimFix	34	1	2.94%
ACS	18	0	0.00%
All	74	4	5.41%

costs 698s, while on-the-fly patch validation via vanilla UniAPR takes only 2.3s to produce the same patch validation results, i.e., 304.89X speedup; when running SimFix on Lang-60, the traditional patch validation costs 924s, while vanilla UniAPR takes only 4s to produce the same patch validation results, i.e., 229.96X speedup; when running CapGen on Math-80, the traditional patch validation costs 18,991s, while vanilla UniAPR takes only 1582s to produce the same patch validation results, i.e., 12.00X speedup. Note that we have further marked various peak speedups in the figure to help better understand the effectiveness of UniAPR. To our knowledge, *this is the first study demonstrating that on-the-fly patch validation can also substantially speed up state-of-the-art source-level APR.*

Precision

We further study the number of bugs that vanilla UniAPR does not produce the same repair results as the traditional patch validation (that restarts a new JVM for each patch). Table 5.4 presents the summarized results for all the studied APR tools on all their fixable bugs. In this table, Column "Tool" presents the studied APR tools, Column "# All" presents the number of all studied fixable bugs for each APR tool, Column "# Mismatch" presents the number of bugs that vanilla UniAPR has inconsistent fixing results with the original APR tool, and Column "Ratio (%)" presents the ratio of bugs with inconsistent results. From this table, we can observe that vanilla UniAPR produces imprecise results for 5.41% of the studied cases overall. To our knowledge, *this is the first empirical study demonstrating that on-the-fly patch validation may produce imprecise/unsound results compared to traditional* *patch validation.* Another interesting finding is that 3 out of the 4 cases with inconsistent patching results occur on the CapGen APR tool. One potential reason is that CapGen is a pattern-based APR system and may generate far more patches than SimFix and ACS. For example, CapGen on average generates over 1,400 patches for each studied bug, while SimFix only generates around 150 on average. In this way, CapGen has way more patches that may affect the correct patch execution than the other studied APR tools. Note that SimFix has only around 150 patches on average since we only studied its fixed bugs; if we had considered all Defects4J bugs studied by the original SimFix paper (including the bugs that cannot be fixed by SimFix), SimFix will produce many more patches, exposing more imprecise/unsound patch validation issues as well as leading to much larger UniAPR speedups.

5.3.2 RQ2: How does on-the-fly patch validation with jvm-reset perform for automated program repair?

Effectiveness

We now present the experimental results for our UniAPR with JVM-reset. The main experimental results are presented in Figure 5.7. In each sub-figure, the horizontal axis presents all the bugs that have been reported to be fixed by each studied tool, while the vertical axis presents the time cost (s); the solid and dashed lines present the time cost for traditional patch validation and UniAPR with JVM reset, respectively. From the figure, we can observe that for all the studied APR tools, UniAPR with JVM reset can also substantially speed up the existing patch validation component with almost no performance degradation. For example, when running ACS on Math-25, the traditional patch validation costs 698s, while on-the-fly patch validation via UniAPR with JVM reset takes only 2.6s to produce the same patch validation results, i.e., 264.47X speedup. Note that we have also marked various peak speedups in the figure to help better understand the effectiveness of UniAPR with JVM reset. While we observe clear speedups for the vast majority of the bugs (and almost no slowdowns), the achieved speedups vary a lot for all the studied APR tools on all the studied bugs. The reason is that the speedups are impacted by many different factors, such as the number of patches executed, the number of bytecode files loaded for each patch execution, the individual test execution time, and so on. For example, we observe that UniAPR even slows down the patch validation for ACS slightly on one bug (i.e., for 1min). Looking into the specific bug (i.e., Math-3), we find that ACS only produces one patch for that bug, and there is no JVM sharing optimization opportunity for UniAPR on-the-fly patch validation. To further confirm our finding, we perform the Pearson Correlation Coefficient analysis [136] between the number of patches for each studied bug and its corresponding speedup for ACS. Shown in Figure 5.9, the horizontal axis denotes the number of patches, while the vertical axis denotes the per-patch speedup (X) achieved; each data point represents one studied bug for ACS. From this figure, we can observe that UniAPR tends to achieve significantly larger speedups for bugs with more patches with a clear positive coefficient R of 0.51 and a p value of 0.031 (which is statistically significant at the significance level of 0.05), demonstrating that *UniAPR with JVM reset can also substantially outperform existing patch validation, with larger speedups for larger systems with more patches.*

Meanwhile, we observe that UniAPR with JVM reset has rather close performance compared with the vanilla UniAPR (shown in Figure 5.6 and Figure 5.7), indicating that UniAPR with JVM reset has negligible overhead compared with the vanilla UniAPR on all the studied bugs for all the studied APR systems. To confirm our finding, Figure 5.8 further presents the time cost comparison among the two UniAPR variants on the three APR systems. In the figure, the horizontal axis presents all the bugs studied for each system while the vertical axis presents the time cost; the solid and dashed lines present the time cost for UniAPR with JVM-reset and vanilla UniAPR, respectively. Shown in the figure, JVM reset has incurred negligible overhead among all the studied bugs for all three systems on UniAPR, e.g., on average 8.33%/7.81%/1.72% overhead for ACS/CapGen/SimFix. The reason is that class reinitializations only need to be performed at certain sites for only the classes with pollution sites. Also, we have various optimizations to speed up JVM reset. For example, although our basic JVM-reset approach in Figure 5.5 performs runtime checks on a `ConcurrentHashMap`, our actual implementation uses arrays for faster class status tracking/check. Furthermore, we observe that the overhead does not change much regardless of the bugs studied, e.g., our UniAPR with JVM-reset has stable overhead across bugs with different number of patches. To further confirm our finding, we perform the Pearson Correlation Coefficient analysis [136] between the number of patches for each studied bug and the corresponding JVM-reset overhead (over vanilla UniAPR) on the ACS tool with the highest overhead. Shown in Figure 5.10, the horizontal axis denotes the number of patches, while the vertical axis denotes the overhead (%) incurred; each data point represents one studied bug for ACS. From this figure, we can observe that there is no clear correlation (at the significance level of 0.05), i.e., JVM-reset overhead is not affected by the numbers of patches. In summary, *UniAPR with JVM-reset only incurs negligible and stable overhead (e.g., less than 8.5% for all studied tools) compared to the vanilla UniAPR, demonstrating the scalability of UniAPR with JVM-reset.*

Precision

According to our experimental results, UniAPR with JVM-rest produces exactly the same APR results as the traditional patch validation, i.e., *UniAPR with JVM-reset successfully fixed all the bugs that vanilla UniAPR failed to fix, mitigating the imprecision/unsoundness of vanilla UniAPR.* We now discuss all 4 bugs that UniAPR with JVM reset can fix while vanilla UniAPR without JVM reset cannot fix:

Figure 5.11 presents the test that fails on the only plausible (also correct) patch of Lang-6 (using CapGen) when running UniAPR without JVM-reset. Given the expected resulting string `‘‘bread &[] butter’’`, the actual returned one is `‘‘bread &[amp;] butter’’`. Digging

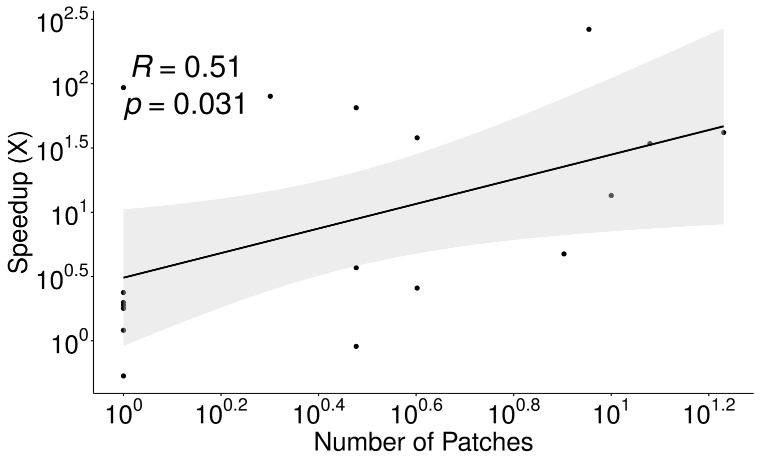


Figure 5.9: Correlation between patch number and speedup achieved by UniAPR with JVM reset

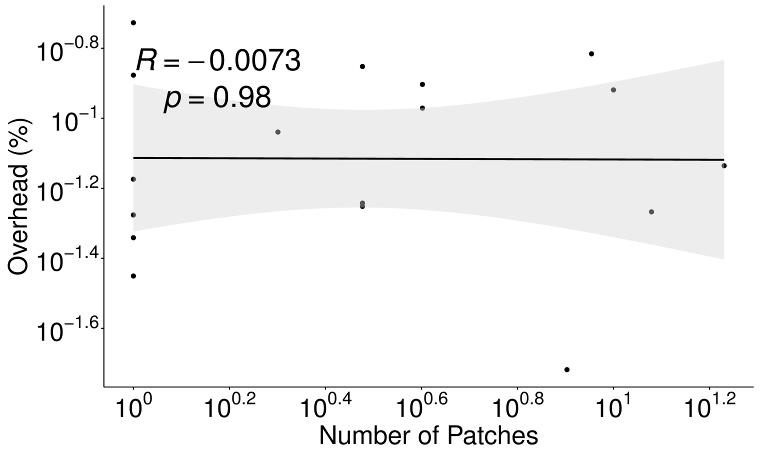


Figure 5.10: Correlation between patch number and overhead incurred by JVM-reset over vanilla UniAPR

```
// org.apache.commons.lang3.StringEscapeUtilsTest.java
    public void testUnescapeHtml4() {
        for (int i = 0; i < HTML_ESCAPES.length; ++i) {
            String message = HTML_ESCAPES[i][0];
            String expected = HTML_ESCAPES[i][2];
            String original = HTML_ESCAPES[i][1];
            // assertion failure: ampersand expected:<bread &[] butter> but was:<bread &[amp;] butter>
            assertEquals(message, expected, StringEscapeUtils.unescapeHtml4(original));
    ...
```

Figure 5.11: Test failed without JVM-reset on Lang-6

into the code, we realize that class `StringEscapeUtils` has a static field named `UNESCAPE_HTML4` , which is responsible for performing the `unescapeHtml4()` method invocation. However, during earlier patch executions, the actual object state of that field is changed, making the `unescapeHtml4()` method invocation return problematic result with vanilla UniAPR. In contrast, when running UniAPR with JVM-reset, field `UNESCAPE_HTML4` will be recreated before each patch execution (if accessed) and will have a clean object state for performing the `unescapeHtml4()` method invocation.

```
// org.apache.commons.math3.EventStateTest.java
   public void testIssue695() {
      FirstOrderDifferentialEquations equation = new FirstOrderDifferentialEquations();
      ...
      double tEnd = integrator.integrate(equation, 0.0, y, target, y);
      ...

   private static class ResettingEvent implements EventHandler {
      private static double lastTriggerTime = Double.NEGATIVE_INFINITY;
      public double g(double t, double[] y) {
         // assertion error
         Assert.assertTrue(t >= lastTriggerTime);
         return t - tEvent;
      }
      ...
```

Figure 5.12: Test failed without JVM-reset on Math-30/41

Figure 5.12 shows another test that fails on the only plausible (and correct) patch of Math-30 when running vanilla UniAPR with CapGen patches, and fails on the only plausible (and correct) patch of Math-41 when running vanilla UniAPR with SimFix patches. Looking into the code, we find that the invocation of `integrate()` in the test will finally call the method `g()` in class `ResettingEvent` (in the bottom). The static field `lastTriggerTime` of class `ResettingEvent` should be `Double.NEGATIVE_INFINITY` in Java, which means the assertion should not fail. Unfortunately, the earlier patch executions pollute the state and change the value of the field. Thus, the test failed when running with vanilla UniAPR on the two plausible patches. In contrast, UniAPR with JVM-reset is able to successfully recover that.

There are four plausible CapGen patches on Math-5 (one is correct) when running with the traditional patch validation. With vanilla UniAPR, all the plausible patches failed on some tests. Figure 5.13 shows the test that fails on three plausible patches (including the correct

```
// org.apache.commons.math3.genetics.UniformCrossoverTest.java
   public class UniformCrossoverTest {
      private static final int LEN = 10000;
      private static final List<Integer> p1 = new ArrayList<Integer>(LEN);
      private static final List<Integer> p2 = new ArrayList<Integer>(LEN);
      public void testCrossover() {
         performCrossover(0.5);
         ...

      private void performCrossover(double ratio) {
         ...
         // assertion failure: expected:<0.5> but was:<5.5095>
         Assert.assertEquals(1.0 - ratio, Double.valueOf((double) from1 / LEN), 0.1);
         ...
```

Figure 5.13: Test failed without JVM-reset on Math-5

```
// org.apache.commons.math3.complex.ComplexTest.java
   public class ComplexTest {
      private double inf = Double.POSITIVE_INFINITY;
      ...
      public void testMultiplyNaNInf() {
         Complex z = new Complex(1,1);
         Complex w = z.multiply(infOne);
         // assertion failure: expected:<-Infinity> but was:<Infinity>
         Assert.assertEquals(w.getReal(), inf, 0);
         ...
```

Figure 5.14: Another test failed without JVM-reset on Math-5

one) on Math-5. The expected value of the assertion should be 0.5, but the actual value turned to 5.5095 due to the change of variable `from1`. After inspecting the code, we found the value of `from1` is decided by two static fields `p1` and `p2` in class `UniformCrossoverTest`. The other earlier patch executions pollute the field values, leading to this test failure when running with vanilla UniAPR. Figure 5.14 presents another test that fails on one plausible patch on Math-5. The expected value from invocation `w.getReal()` should be `Infinity`, which should be the same as field `inf` defined in class `ComplexTest`; however, the actual result from the method invocation is `-Infinity`. The root cause of this test failure is similar to the previous ones, the static fields `NaN` and `INF` in class `Complex` are responsible for the result of method invocation `getReal()`. In this way, `getReal()` returns a problematic result because the earlier patch executions changed the corresponding field values. In contrast, using UniAPR with JVM-reset, all the four plausible patches are successfully produced.

5.4 Discussion

Having single JVM session for validating more than one patch has the immediate benefit of skipping costly JVM restart, reload, and warm-up. As shown by our empirical study, this offers substantial speedups in patch validation. On the other hand, this approach might have the following limitations:

First, the execution of the patches might interfere with each other, i.e., the execution of some tests in one patch might have side-effects affecting the execution of other tests on another patch. UniAPR mitigates these side-effects by resetting static fields to their default values and resetting JDK properties. Although our experimental results demonstrate that such JVM reset can fix all bugs fixed by the traditional patch validation and opens a new dimension for fast&precise patch validation, such in-memory JVM state reset for only class fields might not be sufficient to handle all cases. Also, the side-effects could propagate via operating system or the network. Our current implementation provides a public interface for the users to resolve such issue between patch executions (note that no subject systems in our evaluation require such manual configuration). In the near future, we will study more subject programs to fully investigate the impact of such side effects and design solutions to address them fully automatically.

Second, HotSwap-based patch validation does not support patches that involve changing the layout of the class, e.g. adding/removing fields and/or methods to/from a class. Luckily, the existing APR techniques mainly target patches within ordinary method bodies, and our UniAPR framework is able to reproduce all correct patches for all the three studied state-of-the-art techniques. Another thing worth discussion is that HotSwap originally does not support changes in static initializers; interestingly, our JVM-reset approach can naturally help UniAPR overcome this limitation, since the new initializers can now be reinvoked based on our bytecode transformation to reinitialize the classes. In the near future, we will further look into other promising dynamic class redefinition techniques for implementing our on-the-fly patch validation, such as JRebel [164] and DCEVM [163].

CHAPTER 6
CONCLUSION

Both software testing and debugging are essential steps during software development and after the software is established. However, they can be quite time consuming due to the size and complexity of software systems and it is critical to find some approaches to speed up them. My dissertation is to apply faster software revision testing to decrease the time used for software testing and debugging between different revisions. Applying regression testing to mutation testing and BBI bug detection can largely speed up the process of software testing when systems evolve. Additionally, our unified *on-the-fly* patch validation framework UniAPR can improve the efficiency of patch validation of APR technique which ultimately speeds up software debugging. The contributions of this thesis are as follows. First, we propose the idea of directly applying traditional RTS techniques for incrementally collecting mutation testing results for evolving software systems. We evaluate it on 20 real-world GitHub projects (ranging from 4.31 KLoC to 316.22 KLoC) totalling 1513 revisions and 83.26 Million LoC of code with both state-of-the-art static and dynamic RTS techniques. The experimental results show that surprisingly both file-level static and dynamic RTS can provide rather precise and efficient regression mutation testing supports, while RTS based on finer-grained analysis tends to be imprecise. Secondly, DeBBI can detect library backward incompatibilities using the large number of client project test suites in the wild, i.e., *cross-project* library upgrade testing. We further optimize it via using information retrieval, considering API-use diversity (based on MMR), and test relevance (via extending static RTS) to reduce testing efforts. Our evaluation shows that, compared with the baseline random project prioritization, our approach can reduce the time to detect the first and average unique BBI bug by 99.1% and 70.8% for JDK. Also, we detect 97 real BBI bugs (19 has been confirmed as previously unknown bugs). Finally, we have proposed a unified *on-the-fly* patch validation framework for all

JVM-based APR systems. Compared with the existing on-the-fly patch validation work [69] which only works for bytecode APR, we generalize on-the-fly patch validation to all existing state-of-the-art APR systems at the source code level. We show the first empirical results that on-the-fly patch validation can speed up state-of-the-art representative APR systems, including CapGen, SimFix, and ACS, by over an order of magnitude. Furthermore, we also show the first empirical evidence that on-the-fly patch validation can incur imprecise/unsound patch validation results, and further introduce a new technique for resetting JVM state for precise patch validation with negligible overhead. I believe our works in this dissertation can provide a lot of practical guidance in broad scenarios for automated software testing and debugging.

REFERENCES

[1] (2007). Criticism of windows vista. https://play.google.com/store/apps/details?id=com.sohu.inputmethod.sogou&hl=en. Accessed: 2014-08-30.

[2] (2014). Sougou. https://play.google.com/store/apps/details?id=com.sohu.inputmethod.sogou&hl=en. Accessed: 2014-08-30.

[3] (2016). This library implements hidden markov models (hmm) for time-inhomogeneous markov processes. https://github.com/bmwcarit/hmm-lib.

[4] (2017). Apache camel. http://camel.apache.org/.

[5] (2017). Apache commons math. http://commons.apache.org/proper/commons-math/.

[6] (2017). Apache cxf. http://cxf.apache.org/.

[7] (2017). Ekstazi homepage. http://ekstazi.org/.

[8] (2017). Javalanche mutation testing system. https://github.com/david-schuler/javalanche.

[9] (2017). Major mutation testing system. http://mutation-testing.org/.

[10] (2017). PIT mutation testing system. http://pitest.org/.

[11] (2017). STARTS homepage. https://github.com/TestingResearchIllinois/starts.

[12] (2018). Android Software Development Kit. https://developer.android.com/.

[13] (2018). Apache Struts. https://struts.apache.org/.

[14] (2018). ASM Bytecode Manipulation Framework. http://asm.ow2.org/.

[15] (2018). Implementation of java.util.calendar for the umm al-qura calendar system. https://github.com/msarhan/ummalqura-calendar.

[16] (2018). Indri. http://www.lemurproject.org/indri.php.

[17] (2018). Java Development Kit. http://www.oracle.com/technetwork/java/javase/downloads/index.html/.

[18] (2018). jdeps. https://docs.oracle.com/javase/9/tools/jdeps.htm.

[19] (2018). jdom. http://www.jdom.org/.

[20] (2018). Log4j. http://logging.apache.org/log4j.

[21] (2018). Square Libraries. https://github.com/square.

[22] (2018). The Apache Software Foundation. http://www.apache.org/.

[23] (2019). assertj-core. https://github.com/joel-costigliola/assertj-core/issues/1751.

[24] (2019). Collections-721. https://issues.apache.org/jira/browse/COLLECTIONS-721.

[25] (2019). httpasyncclient. https://issues.apache.org/jira/browse/HTTPASYNC-159.

[26] (2019). java-jwt. https://github.com/auth0/java-jwt/issues/376.

[27] (2019). jena-arq. https://issues.apache.org/jira/browse/JENA-1819.

[28] (2019). Jsoup. https://github.com/jhy/jsoup/issues/1274.

[29] (2019). lombok. https://github.com/rzwitserloot/lombok/issues/2320.

[30] (2019). mybatis-spring. https://github.com/mybatis/spring/issues/427.

[31] (2019). reflection. https://github.com/ronmamo/reflections/issues/277.

[32] (2019). vfs. https://issues.apache.org/jira/browse/VFS-7392320.

[33] aaa. Apache Maven. https://maven.apache.org/.

[34] aaa. Joda-Time. https://github.com/JodaOrg/joda-time.

[35] Acree, A. T., T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward (1979). Mutation analysis. Technical report, GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE.

[36] Ahmed, I., C. Jensen, A. Groce, and P. E. McKenney (2017). Applying mutation analysis on kernel test suites: An experience report. In *ICSTW*, pp. 110–115. IEEE.

[37] Ammann, P. and J. Offutt (2016). *Introduction to software testing*. Cambridge University Press.

[38] Andrews, J. H., L. C. Briand, and Y. Labiche (2005). Is mutation an appropriate tool for testing experiments? In *ICSE*, pp. 402–411. ACM.

[39] Asuncion, H. U., A. U. Asuncion, and R. N. Taylor (2010). Software traceability with topic modeling. In *Proceedings of the 32nd ACM/IEEE international conference on Software Engineering-Volume 1*, pp. 95–104. ACM.

[40] Balaban, I., F. Tip, and R. Fuhrer (2005). Refactoring support for class library migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 265–279.

[41] Barr, E. T., Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro (2014). The plastic surgery hypothesis. In *FSE*, FSE'14, pp. 306–317.

[42] Bell, J. and G. Kaiser (2014). Unit test virtualization with vmvm. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 550–561.

[43] Benton, S., X. Li, Y. Lou, and L. Zhang (2020). On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 907–918.

[44] Blei, D. M., A. Y. Ng, and M. I. Jordan (2003). Latent dirichlet allocation. *Journal of machine Learning research 3*(Jan), 993–1022.

[45] Boulder, C. (2013). University of cambridge study: Failure to adopt reverse debugging costs global economy $41 billion annually. `https://bit.ly/2T4fWnq`. Accessed: August, 2020.

[46] Bozkurt, M. and M. Harman (2011, Dec). Automatically generating realistic test input from web services. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*, pp. 13–24.

[47] Bryce, R. C. and A. M. Memon (2007). 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, New York, NY, USA, pp. 1–7. ACM.

[48] Carbonell, J. and J. Goldstein (1998). The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 335–336. ACM.

[49] Chen, L., F. Hassan, X. Wang, and L. Zhang (2020). Taming behavioral backward incompatibilities via cross-project testing and analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 112–124.

[50] Chen, L., Y. Ouyang, and L. Zhang (2021). Fast and precise on-the-fly patch validation for all. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1123–1134.

[51] Chen, L., Y. Pei, and C. A. Furia (2017). Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 637–647. IEEE.

[52] Chen, L. and L. Zhang (2018). Speeding up mutation testing via regression test selection: An extensive study. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 58–69.

[53] Chow, K. and D. Notkin (1996). Semi-automatic update of applications in response to library changes. In *Software Maintenance 1996, Proceedings., International Conference on*, pp. 359–368.

[54] Coles, H., T. Laurent, C. Henard, M. Papadakis, and A. Ventresque (2016). Pit: a practical mutation testing tool for java. In *ISSTA*, pp. 449–452. ACM.

[55] Corporation, O. (2020). Java instrumentation api. Accessed: August, 2020.

[56] Cossette, B. and R. J. Walker (2012). Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, pp. 55.

[57] D Le, X. B. (2018). *Overfitting in Automated Program Repair: Challenges and Solutions*. Ph. D. thesis, Singapore Management University.

[58] Debroy, V. and W. E. Wong (2010). Using mutation to automatically suggest fixes for faulty programs. In *ICST*, pp. 65–74. IEEE.

[59] DeMillo, R. A., R. J. Lipton, and F. G. Sayward (1978). Hints on test data selection: Help for the practicing programmer. *Computer 11*(4), 34–41.

[60] Dig, D. and R. Johnson (2005). The role of refactorings in api evolution. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 389–398.

[61] Do, H. and G. Rothermel (2006). On the use of mutation faults in empirical assessments of test case prioritization techniques. *TSE 32*(9), 733–752.

[62] Do, H., G. Rothermel, and A. Kinneer (2004). Empirical studies of test case prioritization in a junit testing environment. In *ISSRE*, pp. 113–124.

[63] Durieux, T. and M. Monperrus (2016). Dynamoth: dynamic code synthesis for automatic program repair. In *AST*, pp. 85–91.

[64] Elbaum, S., A. G. Malishevsky, and G. Rothermel (2000). Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, New York, NY, USA, pp. 102–112. ACM.

[65] Elbaum, S., G. Rothermel, and J. Penix (2014). Techniques for improving regression testing in continuous integration development environments. In *FSE*, pp. 235–245. ACM.

[66] Frankl, P. G., S. N. Weiss, and C. Hu (1997). All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software 38*(3), 235–253.

[67] Fraser, G. and A. Zeller (2012). Mutation-driven generation of unit tests and oracles. *TSE 38*(2), 278–292.

[68] Gazzola, L., D. Micucci, and L. Mariani (2017). Automatic software repair: A survey. *IEEE Transactions on Software Engineering 45*(1), 34–67.

[69] Ghanbari, A., S. Benton, and L. Zhang (2019). Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 19–30.

[70] Gligoric, M., L. Eloussi, and D. Marinov (2015). Practical regression test selection with dynamic file dependencies. In *ISSTA*, pp. 211–222. ACM.

[71] Goldstein, J., V. Mittal, J. Carbonell, and M. Kantrowitz (2000). Multi-document summarization by sentence extraction. In *Proceedings of the 2000 NAACL-ANLP Workshop on Automatic summarization*, pp. 40–48. Association for Computational Linguistics.

[72] Gopinath, R., C. Jensen, and A. Groce (2014). Code coverage for suite evaluation by developers. In *ICSE*, pp. 72–82. ACM.

[73] Goues, C. L., M. Pradel, and A. Roychoudhury (2019). Automated program repair. *Communications of the ACM 62*(12), 56–65.

[74] Groce, A., I. Ahmed, C. Jensen, and P. E. McKenney (2015). How verified is my code? falsification-driven verification (t). In *ASE*, pp. 737–748. IEEE.

[75] Guo, S. and S. Sanner (2010). Probabilistic latent maximal marginal relevance. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pp. 833–834. ACM.

[76] Hamlet, R. G. (1977). Testing programs with the aid of a compiler. *TSE* (4), 279–290.

[77] Harrold, M. J., J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi (2001). Regression test selection for java software. In *ACM SIGPLAN Notices*, Volume 36, pp. 312–326. ACM.

[78] Hiemstra, D. (2001). Using language models for information retrieval.

[79] Hong, L., O. Dan, and B. D. Davison (2011). Predicting popular messages in twitter. In *Proceedings of the 20th international conference companion on World wide web*, pp. 57–58. ACM.

[80] Howden, W. E. (1982). Weak mutation testing and completeness of test sets. *TSE* (4), 371–379.

[81] Hua, J., M. Zhang, K. Wang, and S. Khurshid (2018). Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 12–23.

[82] Jia, Y. and M. Harman (2011). An analysis and survey of the development of mutation testing. *TSE 37*(5), 649–678.

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

[84] Jiang, J., L. Ren, Y. Xiong, and L. Zhang (2019). Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 255–266.

[85] Jiang, J., Y. Xiong, H. Zhang, Q. Gao, and X. Chen (2018). Shaping program repair space with existing patches and similar code. In *ISSTA*, pp. 298–309. ACM.

[86] Just, R., D. Jalali, and M. D. Ernst (2014). Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440.

[87] Just, R., D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser (2014). Are mutants a valid substitute for real faults in software testing? In *FSE*, pp. 654–665. ACM.

[88] Kim, D., J. Nam, J. Song, and S. Kim (2013). Automatic patch generation learned from human-written patches. In *ICSE*, pp. 802–811. IEEE Press.

[89] Kim, S., Y. Song, K. Kim, J.-W. Cha, and G. G. Lee (2006). Mmr-based active machine learning for bio named entity recognition. In *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*, pp. 69–72. Association for Computational Linguistics.

[90] Kit, E. and S. Finzi (1995). *Software testing in the real world: improving the process.* ACM Press/Addison-Wesley Publishing Co.

[91] Korel, B., L. H. Tahat, and M. Harman (2005). Test prioritization using system models. In *ICSM*, pp. 559–568.

[92] Krestel, R., P. Fankhauser, and W. Nejdl (2009). Latent dirichlet allocation for tag recommendation. In *Proceedings of the third ACM conference on Recommender systems*, pp. 61–68. ACM.

[93] Kung, D. C., J. Gao, P. Hsia, J. Lin, and Y. Toyoshima (1995). Class firewall, test order, and regression testing of object-oriented programs. *JOOP 8*(2), 51–65.

[94] Landauer, T. K., P. W. Foltz, and D. Laham (1998). An introduction to latent semantic analysis. *Discourse processes 25*(2-3), 259–284.

[95] Le Goues, C., T. Nguyen, S. Forrest, and W. Weimer (2012). Genprog: A generic method for automatic software repair. *IEEE TSE 38*(1), 54–72.

[96] Lee, C. and G. G. Lee (2006). Information gain and divergence-based feature selection for machine learning-based text categorization. *Information processing & management 42*(1), 155–165.

[97] Lee, D. L., H. Chuang, and K. Seamons (1997). Document ranking and the vector-space model. *IEEE software 14*(2), 67–75.

[98] Legunsen, O., F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov (2016). An extensive study of static regression test selection in modern software evolution. In *FSE*, pp. 583–594. ACM.

[99] Leung, H. K. and L. White (1990). A study of integration testing and software regression at the integration level. In *ICSM*, pp. 290–301. IEEE.

[100] Lezama, A. S. (2008). *Program synthesis by sketching.* Ph. D. thesis, PhD thesis, EECS Department, University of California, Berkeley.

[101] Li, X., W. Li, Y. Zhang, and L. Zhang (2019). Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 169–180.

[102] Li, X. and L. Zhang (2017). Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages 1*(OOPSLA), 92.

[103] Li, Z., M. Harman, and R. M. Hierons (2007). Search algorithms for regression test case prioritization. *IEEE Trans. Software Eng. 33*(4), 225–237.

[104] Liu, K., A. Koyuncu, D. Kim, and T. F. Bissyandé (2019). Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 31–42.

[105] Liu, K., S. Wang, A. Koyuncu, K. Kim, T. F. D. A. Bissyande, D. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon (2020). On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *42nd ACM/IEEE International Conference on Software Engineering (ICSE).*

[106] Liu, X. and H. Zhong (2018). Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 118–129. IEEE.

[107] Long, F. and M. Rinard (2015). Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 166–178.

[108] Long, F. and M. Rinard (2016). Automatic patch generation by learning correct code. In *POPL*, POPL'16, pp. 298–312.

[109] Lou, Y., J. Chen, L. Zhang, D. Hao, and L. Zhang (2019). History-driven build failure fixing: how far are we? In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 43–54.

[110] Lou, Y., A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang (2020). Can automated program repair refine fault localization? a unified debugging approach. In *ISSTA*. to appear.

[111] Lu, Y., Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang (2016). How does regression test prioritization perform in real-world software evolution? In *ICSE*, pp. 535–546.

[112] Lucia, A. D., F. Fasano, R. Oliveto, and G. Tortora (2007). Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM) 16*(4), 13.

[113] Ma, L., C. Zhang, B. Yu, and J. Zhao (2016). Retrofitting automatic testing through library tests reusing. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–4.

[114] Maarek, Y. S., D. M. Berry, and G. E. Kaiser (1991). An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on software Engineering 17*(8), 800–813.

[115] Martinez, M. and M. Monperrus (2016). Astor: A program repair library for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pp. 441–444.

[116] McDonnell, T., B. Ray, and M. Kim (2013). An empirical study of api stability and adoption in the android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pp. 70–79.

[117] Mechtaev, S., J. Yi, and A. Roychoudhury (2016). Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pp. 691–701.

[118] Mehne, B., H. Yoshida, M. R. Prasad, K. Sen, D. Gopinath, and S. Khurshid (2018). Accelerating search-based program repair. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 227–238. IEEE.

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

[120] Monperrus, M. (2018). Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR) 51*(1), 1–24.

[121] Monperrus, M. (2020). The living review on automated program repair. Technical report.

[122] Monperrus, M., S. Urli, T. Durieux, M. Martinez, B. Baudry, and L. Seinturier (2019, July). Repairnator patches programs automatically. *Ubiquity 2019*(July).

[123] Moon, S., Y. Kim, M. Kim, and S. Yoo (2014). Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, pp. 153–162. IEEE.

[124] Mostafa, S., R. Rodriguez, and X. Wang (2015). A study on behavioral backward incompatibilities of java software libraries. In *Proceedings of the 2017 International Symposium on Software Testing and Analysis*, pp. 215–225.

[125] Myers, G. J., C. Sandler, and T. Badgett (2011). *The art of software testing*. John Wiley & Sons.

[126] Nguyen, H. D. T., D. Qi, A. Roychoudhury, and S. Chandra (2013). Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 772–781. IEEE.

[127] Nielson, F. and H. R. Nielson (2019). *Formal Methods*. Springer.

[128] Objectweb (2020). Asm bytecode manipulation framework. Accessed: August, 2020.

[129] Offutt, A. J., A. Lee, G. Rothermel, R. H. Untch, and C. Zapf (1996). An experimental determination of sufficient mutant operators. *TOSEM 5*(2), 99–118.

[130] Oracle (2020). The java language specification. Accessed: August, 2020.

[131] Orso, A., N. Shi, and M. J. Harrold (2004). Scaling regression testing to large software systems. In *ACM SIGSOFT Software Engineering Notes*, Volume 29, pp. 241–251. ACM.

[132] Oster, N. and F. Saglietti (2006). Automatic test data generation by multi-objective optimisation. In *International Conference on Computer Safety, Reliability, and Security*, pp. 426–438. Springer.

[133] Ounis, I., G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma (2006). Terrier: A high performance and scalable information retrieval platform. In *Proceedings of the OSIR Workshop*, pp. 18–25.

[134] Papadakis, M. and Y. Le Traon (2012). Using mutants to locate" unknown" faults. In *ICST*, pp. 691–700. IEEE.

[135] Papadakis, M. and N. Malevris (2010). Automatic mutation test case generation via dynamic symbolic execution. In *ISSRE*, pp. 121–130. IEEE.

[136] Pearson, K. (1895). Notes on regression and inheritance in the case of two parents proceedings of the royal society of london, 58, 240-242.

[137] Porteous, I., D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling (2008). Fast collapsed gibbs sampling for latent dirichlet allocation. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 569–577. ACM.

[138] Porter, M. F. (2001). Snowball: A language for stemming algorithms.

[139] Pradel, M. and T. R. Gross (2012). Leveraging test generation and specification mining for automated bug detection without false positives. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, Piscataway, NJ, USA, pp. 288–298. IEEE Press.

[140] Qi, Z., F. Long, S. Achour, and M. Rinard (2015). An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, pp. 24–36. ACM.

[141] Raemaekers, S., A. van Deursen, and J. Visser (2017, July). Semantic versioning and impact of breaking changes in the maven repository. *J. Syst. Softw. 129*(C), 140–158.

[142] Raftery, A. E. and S. Lewis (1991). How many iterations in the gibbs sampler? Technical report, WASHINGTON UNIV SEATTLE DEPT OF STATISTICS.

[143] Ramler, R. and K. Wolfmaier (2006). Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test*, pp. 85–91.

[144] Reiss, S. P. (2014). Towards creating test cases using code search. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pp. 436–440.

[145] Ren, X., F. Shah, F. Tip, B. G. Ryder, and O. Chesley (2004). Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, Volume 39, pp. 432–448. ACM.

[146] Ren, X., F. Shah, F. Tip, B. G. Ryder, O. Chesley, and J. Dolby (2003). Chianti: A prototype change impact analysis tool for java. *Rutgers University, Department of Computer Science, Tech. Rep. DCS-TR-533*.

[147] Robertson, S., H. Zaragoza, and M. Taylor (2004). Simple bm25 extension to multiple weighted fields. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pp. 42–49. ACM.

[148] Robertson, S. E., S. Walker, S. Jones, M. M. Hancock-Beaulieu, M. Gatford, et al. (1995). Okapi at trec-3. *Nist Special Publication Sp 109*, 109.

[149] Rothenberg, B.-C. and O. Grumberg (2016). Sound and complete mutation-based program repair. In *Proc. of FM*, pp. 593–611. Springer.

[150] Rothermel, G. and M. J. Harrold (1996). Analyzing regression test selection techniques. *IEEE Transactions on software engineering 22*(8), 529–551.

[151] Rothermel, G. and M. J. Harrold (1997). A safe, efficient regression test selection technique. *TOSEM 6*(2), 173–210.

[152] Rothermel, G., R. H. Untch, C. Chu, and M. J. Harrold (1999). Test case prioritization: An empirical study. In *ICSM*, pp. 179–188.

[153] Ryder, B. G. and F. Tip (2001). Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 46–53. ACM.

[154] Saha, R. K., M. Lease, S. Khurshid, and D. E. Perry (2013). Improving bug localization using structured information retrieval. In *ASE*, pp. 345–355.

[155] Saha, R. K., L. Zhang, S. Khurshid, and D. E. Perry (2015, May). An information retrieval approach for regression test prioritization based on program changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Volume 1, pp. 268–279.

[156] Salton, G., A. Wong, and C.-S. Yang (1975). A vector space model for automatic indexing. *Communications of the ACM 18*(11), 613–620.

[157] Schuler, D. and A. Zeller (2009). Javalanche: efficient mutation testing for java. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 297–298. ACM.

[158] Shen, Y., X. He, J. Gao, L. Deng, and G. Mesnil (2014). A latent semantic model with convolutional-pooling structure for information retrieval. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pp. 101–110. ACM.

[159] Shi, A., A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov (2014). Balancing trade-offs in test-suite reduction. In *FSE*, pp. 246–256. ACM.

[160] Shi, A., T. Yung, A. Gyori, and D. Marinov (2015). Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 237–247. ACM.

[161] Siami Namin, A., J. H. Andrews, and D. J. Murdoch (2008). Sufficient mutation operators for measuring test effectiveness. In *ICSE*, pp. 351–360. ACM.

[162] Software, U. (2016). Increasing software development productivity with reversible debugging. `https://bit.ly/3c8ccbn`. Accessed: August, 2020.

[163] Team, D. (2020a). Dynamic code evolution vm for java. Accessed: August, 2020.

[164] Team, J. (2020b). Jrebel. Accessed: August, 2020.

[165] team, U. (2020). Uniapr webpage. Accessed: August, 2020.

[166] Thomas, S. W. (2011). Mining software repositories using topic models. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 1138–1139. ACM.

[167] Thummalapenta, S., J. de Halleux, N. Tillmann, and S. Wadsworth (2010). Dygen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In G. Fraser and A. Gargantini (Eds.), *Tests and Proofs*, Berlin, Heidelberg, pp. 77–93. Springer Berlin Heidelberg.

[168] Thummalapenta, S., T. Xie, N. Tillmann, J. de Halleux, and W. Schulte (2009). Mseqgen: Object-oriented unit-test generation via mining source code. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, New York, NY, USA, pp. 193–202. ACM.

[169] Thummalapenta, S., T. Xie, N. Tillmann, J. de Halleux, and Z. Su (2011). Synthesizing method sequences for high-coverage testing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, New York, NY, USA, pp. 189–206. ACM.

[170] Tian, K., M. Revelle, and D. Poshyvanyk (2009). Using latent dirichlet allocation for automatic categorization of software. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pp. 163–166. IEEE.

[171] Tian, Y., D. Lo, and C. Sun (2012). Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pp. 215–224. IEEE.

[172] Timperley, C. S., S. Stepney, and C. Le Goues (2017). An investigation into the use of mutation analysis for automated program repair. In *SSBSE*, pp. 99–114. Springer.

[173] Voorhees, E. M., D. K. Harman, et al. (2005). *TREC: Experiment and evaluation in information retrieval*, Volume 1. MIT press Cambridge.

[174] Wang, X. and E. Grimson (2008). Spatial latent dirichlet allocation. In *Advances in neural information processing systems*, pp. 1577–1584.

[175] Wen, M., J. Chen, R. Wu, D. Hao, and S.-C. Cheung (2018). Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 1–11. IEEE.

[176] Wong, W. E., R. Gao, Y. Li, R. Abreu, and F. Wotawa (2016, August). A survey on software fault localization. *IEEE TSE 42*(8), 707–740.

[177] Wong, W. E. and A. P. Mathur (1995). Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software 31*(3), 185–196.

[178] Woodward, M. and K. Halewood (1988). From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pp. 152–158. IEEE.

[179] Xiong, Y., J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang (2017). Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, pp. 416–426.

[180] Xuan, J., M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus (2017). Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering 43*(1), 34–55.

[181] Zhai, C. and J. Lafferty (2004). A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems (TOIS) 22*(2), 179–214.

[182] Zhang, J., Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang (2016). Predictive mutation testing. In *Proc. of ISSTA*, pp. 342–353. ACM.

[183] Zhang, L., M. Gligoric, D. Marinov, and S. Khurshid (2013). Operator-based and random mutant selection: Better together. In *ASE*, pp. 92–102.

[184] Zhang, L., D. Hao, L. Zhang, G. Rothermel, and H. Mei (2013). Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE*, pp. 192–201.

[185] Zhang, L., S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei (2010). Is operator-based mutant selection superior to random mutant selection? In *ICSE*, Volume 1, pp. 435–444. IEEE.

[186] Zhang, L., M. Kim, and S. Khurshid (2011). Localizing failure-inducing program edits based on spectrum information. In *ICSM*, pp. 23–32. IEEE.

[187] Zhang, L., M. Kim, and S. Khurshid (2012). Faulttracer: a change impact and regression fault analysis tool for evolving java programs. In *FSE*, pp. 40. ACM.

[188] Zhang, L., D. Marinov, L. Zhang, and S. Khurshid (2012). Regression mutation testing. In *ISSTA*, pp. 331–341. ACM.

[189] Zhang, L., T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei (2010). Test generation via dynamic symbolic execution for mutation testing. In *ICSME*, pp. 1–10.

[190] Zhang, L., L. Zhang, and S. Khurshid (2013). Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*, Volume 48, pp. 765–784. ACM.

[191] Zhou, J., H. Zhang, and D. Lo (2012). Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pp. 14–24.

BIOGRAPHICAL SKETCH

Lingchao is a PhD candidate from The University of Texas at Dallas who majored in Computer Science. His main research interests lie in Software Engineering, especially Software Testing and Software Debugging. He is dedicated to developing practical software testing, analysis, and debugging systems to predict, detect, and fix bugs for all kinds of software systems. He has published some research papers in top-tier conferences and journals on speeding up software testing and debugging.

CURRICULUM VITAE

Lingchao Chen

October 22, 2021

Contact Information:

Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: `lingchao.chen@utdallas.edu`

Educational History:

BS, Computer Science, University of Electronic Science and Technology of China, Chengdu, China, 2016
PhD, Computer Science, The University of Texas at Dallas, 2022

Publications:

1. **Lingchao Chen** and Lingming Zhang. Speeding up Mutation Testing via Regression Test Selection: An Extensive Study. In *proceedings of the IEEE Conference on Software Testing, Validation and Verification (ICST 2018)*, April 2018.
2. Dongyu Mao, **Lingchao Chen**, and Lingming Zhang. An Extensive Study on Cross-project Predictive Mutation Testing. In *proceedings of the IEEE Conference on Software Testing, Validation and Verification (ICST 2019)*, April 2019.
3. Mengshi Zhang, Yaoxian Li, Xia Li, **Lingchao Chen**, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. An Empirical Study of Boosting Spectrum-based Fault Localization via PageRank. In *proceedings of the IEEE Transactions on Software Engineering (TSE).*
4. **Lingchao Chen**, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. Taming Behavioral Backward Incompatibilities via Cross-Project Testing and Analysis. In *proceedings of the 42nd IEEE/ACM International Conference on Software Engineering (ICSE 2020)*, May 2020.
5. **Lingchao Chen**, Yichen Ouyang, and Lingming Zhang. Fast and Precise On-the-fly Patch Validation for All. In *proceedings of the 42nd IEEE/ACM International Conference on Software Engineering (ICSE 2021)*, May 2021.