AMERICAN UNIVERSITY OF BEIRUT

AUTOMATED PROGRAM REPAIR USING FLOW ALTERATION

by
CHADI HANNA TRAD

A dissertation
submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
to the Department of Electrical and Computer Engineering
of the Maroun Semaan Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
May 2021

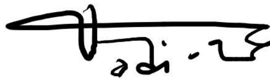
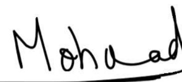
AMERICAN UNIVERSITY OF BEIRUT

AUTOMATED PROGRAM REPAIR USING FLOW ALTERATION

by

CHADI HANNA TRAD

Approved by:

Dr. Louay Bazzi, Professor Electrical and Computer Engineering	Chairman
Dr. Wassim Masri, Professor Electrical and Computer Engineering	Advisor
Dr. Fadi Zaraket, Associate Professor Electrical and Computer Engineering	Member of Committee 
Dr. Mohamad Jaber Google	Member of Committee 
Dr. Andy Podgurski, Professor Case Western Reserve University	Member of Committee
Dr. Rawad Abou Assi, Assistant Professor Electrical and Computer Engineering, Lebanese International University	Member of Committee

Date of dissertation defense: April 27, 2021

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT,
RELEASE FORM

Student Name: Trad ______________ Chadi ______________ Hanna
Last First Middle

◯ Master's Thesis ◯ Master's Project ✓ Doctoral Dissertation

☐ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

■ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after:

One ___ year from the date of submission of my thesis, dissertation or project.
Two ___ years from the date of submission of my thesis , dissertation or project.
Three ✓ years from the date of submission of my thesis , dissertation or project.

2021-05-10

Signature Date

Acknowledgements

First and foremost, I heartfully thank my thesis advisor, Prof. Wassim Masri, for his constant support and continued trust throughout my Ph.D. studies. His good advice and assistance go beyond my thesis to cover a wide variety of topics including software practices and career paths. I am also thankful for being part of a dedicated research group with Prof. Wassim and Prof. Rawad, and forever grateful for all the lessons learned, which will serve me well for many years to come. I would like to thank the committee members for offering invaluable comments and directions and for putting this thesis on the right track.

I am greatly thankful for all the memories that AUB provided me, and all the lessons I learned in this institution. In particular, most of my interests and skills in software were influenced and shaped by Prof. Louay Bazzi, Prof. Fadi Zaraket, and Prof. Wassim Masri. They each provided a unique perspective and depth to this domain and were always available for advice and discussions.

I dedicate this thesis to my parents, to whom I owe it all, and I thank Farah Saab for her help and for nudging me in the right direction.

Finally, I would like to acknowledge AUB and the National Council for Scientific Research of Lebanon (CNRS-L) for granting me a doctoral fellowship.

An Abstract of the Dissertation of

Chadi Hanna Trad for Doctor of Philosophy
Major: Electrical and Computer Engineering

Title: Automated Program Repair Using Flow Alteration

Program bugs are notorious for their cost when discovered late in the Software Development Life Cycle (SDLC). When a bug is discovered in the production environment, the cycle is restarted, which often takes at least one or two weeks before the fix is pushed to production. To mitigate the increasing cost of delayed bug resolution, several families of techniques were developed such as Automated Program Repair (APR), and Failure Detection. In this thesis, we propose two variants of these techniques. To perform a program repair, most approaches apply methods based on Satisfiability Modulo Theorem (SMT) or on Generate-and-Validate (G&V) techniques. However, these approaches are computationally complex. SMT-based methods can suffer from path explosion for large programs and may require the programmer to write assertions and specifications, while G&V methods tend to have a huge search-space and may over-fit the test suite. Our first proposed method, *CFAAR*, is a test-based repair technique that operates by selectively altering the outcome of suspicious control statements in order to yield the expected program behavior. *CFAAR* targets defects that are repairable by altering the execution of control statements under specific conditions. Unlike other test-based repair techniques that mine for patches in other parts of the program or in various artifacts, *CFAAR* relies on the program's state to determine when a candidate control statement should be negated to yield a correct behavior. Then, the captured state is further analyzed to synthesize a patch in the form of a conditional that guards the candidate control statement. Our second proposed method, *D-FUSE*, is a failure detection method that uses both *structural* and *substate* profiles. While *structural* profiles estimate the various path components visited in each run, *substate* profiles characterize the various values taken by a variable for a given run. Furthermore, a *substate* descriptor embeds statistical

information about these values, and how they cluster among all test cases. In this thesis, we leverage both types of profiles to enhance failure detection. After showing that the two profile types are complementary for detecting failure, we propose an optimization technique that selects a set of profile elements that predicts failure, while minimizing the profiling cost. Finally, an augmented set of instrumentation probes are selected to reproduce the selected profile elements, while maintaining a low resource overhead.

Contents

Acknowledgements **v**

Abstract **vi**

1 Introduction **1**

1.1 CFAAR: An Automated Program Repair technique 2

1.2 D-FUSE: A Failure Detection Technique 5

1.3 Dissertation Outline 7

2 Control Flow Alteration to Assist Repair **8**

2.1 Overview 9

2.2 CFAAR: Selective Control Flow Alteration 11

2.2.1 Overview 12

2.2.2 Heuristic CFA Search 13

2.2.3 Building the Classifiers 15

2.3 CBFL: Identifying Failure-Causing Predicates 21

2.3.1 Causal Inference: Background 22

2.3.2 From Failure-Correlated to Failure-Causing 24

2.4 Empirical Results 25

2.5 Subject Programs and Test Suites 25

2.6 RQ1: How Prevalent are the Defects that are Potentially Repairable by CFAAR? . . . 26
2.7 RQ2: How Effective is CFAAR at Synthesizing Plausible Patches? 28
2.8 RQ3: How Effective is CFAAR at Synthesizing Correct Patches? . 31
2.9 Threats to Validity . . . 32
2.10 Analysis . . . 33

3 Detecting Failure Using Substate Profile Elements 34
3.1 Background . . . 36
3.1.1 Structural Profiles . . . 36
3.1.2 Substate Profiles . . . 37
3.2 Motivating Example . . . 38
3.3 Empirical Study . . . 40
3.4 Online Failure Detection . . . 42
3.4.1 Architecture . . . 43
3.4.2 Experimental Setup and Results . . . 50
3.5 Analysis . . . 58

4 Related Work 59
4.1 Related to Repair . . . 59
4.2 Failure Detection as an Intrusion Detection System . . . 64

5 Conclusion 67

A Complementary Material for the Repair Method 69
A.1 Analysis of the Patches Generated on Suspicious Predicates . . . 70
A.1.1 Sample Correct Patch 1: print_tokens2 - v8 . . . 70
A.1.2 Sample Correct Patch 2: syllables - v1 (v2 is very similar) 71

A.1.3 Sample Correct Patch 3: grade - v13 72
A.1.4 Sample Correct Patch 4: digits- v19 73
A.1.5 Sample Incorrect Patch - tcas - v1 74
A.1.6 Sample Undecided Patch 75
A.2 Patches generated on manually specified predicates 76

Bibliography **80**

List of Figures

2.1 CFAAR Overview 12

2.2 Heuristic CFA Search Algorithm 14

2.3 Decision Tree Conversion Example 18

2.4 Instrumentation for Negating Predicates 19

3.1 Generation of failure detection rules. After profiling all subject program runs, clustering is performed, and each run gets associated with the clusters it covers. Finally, an optimization engine determines which combination of clusters can predict. 44

3.2 Generating the modified subject program. We extract a set of simple profiling elements that can help generate these statistics. The information about the profiling elements and the rules to be used for detection are passed to the instrumenter, which in turn generates a modified subject program. 44

3.3 Modified subject program with failure detection. The original subject program logic is used, with two additional modifications: (1) probes are inserted at selected locations, and (2) before exiting, the program checks if failure detection rules are met. 45

List of Tables

2.1 Information about Subject Programs 25

2.2 Summary of Results 27

2.3 Comparison to state-of-the-art results 31

3.1 A sample of the features generated by one capture point 38

3.2 Substate profiles and failure detection code 39

3.3 Number of versions having at least one profile element associated with failure 41

3.4 Table illustrating when AND can be used to detect failure 48

3.5 Table illustrating when OR can be used to detect failure 49

3.6 Failure detection accuracy for different profile types 52

3.7 Distribution of the number of profile elements to be combined with the OR operation 54

3.8 Range of time taken during the optimization process per program version (in minutes) 55

3.9 Average memory used in megabytes during the optimization process 55

A.1 Summary of the patches generated for print_tokens, print_tokens2, schedule, schedule2 77

A.2 Summary of the patches generated for tot_info 78

A.3 Summary of the patches generated for tcas and replace 79

Chapter 1

Introduction

Software defects are abundant. On a large scale, the cost of repairing bugs in 2013 was 312 billion dollars, according to a report published in [1]. Debugging, testing, and verification can add up to around 75% of the cost of a software development project [2]. On a smaller scale, software developers spend around 80% of their time finding and fixing bugs. Due to the time-consuming nature of manual fault localization and repair, it is common practice to have software products shipped with pre-identified low-priority bugs. While some defects can be harmless, others can cause applications to crash, hundred-million-dollar space missions to fail, and death in some cases [3]. On the other hand, a long Software Development Life Cycle (SDLC) can often result in increasing costs before a vulnerability is fixed. Due to all of these factors and more, there is a pressing need for automated tools to help identify, localize, repair, or mitigate the costs of software defects.

According to the PIE model [4], a failure occurs when a defect is **e**xecuted, when the state of the program is **i**nfected and when this infection propagates to the output of the **p**rogram. Our thesis studies and enhances two failure mitigation techniques. The first technique that we study is Automated Program Repair

(APR). For a given defect, an APR technique characterizes the bug and generates a fix such that the program produces the expected output for both the passing and the failing runs. The second technique that we study is Failure Detection. With Failure Detection, a signal can be raised to invalidate or revert the transaction and potentially signal operators and developers about the execution of the vulnerability. Each of the studied techniques can be used in its own context, depending on the vulnerability type, severity, and the system architecture.

The next two sections detail our contribution to each of the proposed techniques. In the last section, we outline the rest of the dissertation.

1.1 CFAAR: An Automated Program Repair technique

APR investigates the repair of software by either modifying its source code or bytecode (known as Program Repair) or by fixing its state (known as State Repair). While the purpose of the former method is to correct a software on the source-code level, the purpose of latter is to allow continued execution when run-time errors occur. Recent techniques have shown significant progress in Program Repair, using evolutionary techniques, and specification-based synthesis. Evolutionary techniques focus on modifying the program randomly with the purpose of making all test cases pass. On the other hand, specification-based synthesis make use of developer-written specifications to guide the repair process.

Most approaches apply computationally expensive methods. Specification-based methods can suffer from path explosion on large programs and may require the programmer to write assertions, while evolutionary methods tend to have a huge search-space and often overfit the test suite. In this thesis, we focus on us-

ing a combination of machine-learning, instrumentation and run-time monitoring techniques to guide test-case-driven techniques in program repair.

In the remainder of this section, we present an overview of our work published in [5]. We propose an automated repair technique that functions by Control Flow Alteration to Assist Repair (*CFAAR*). *CFAAR* is a test-based repair technique that operates by selectively altering the outcome of suspicious control statements in order to yield expected program behavior, and subsequently provide a synthesized patch that can be used directly or assist the programmer in the repair process. It focuses on the category of defects that are repairable by negating control statements under specific conditions. Unlike many test-based repair techniques that mine for patches in other parts of the program [6, 7] or in various artifacts, *CFAAR* relies on the program's state to determine when a candidate control statement should be negated to make failures pass. The captured state information is further analyzed to synthesize a patch in the form of a conditional that guards the candidate control statement. Specifically, given a test suite in which the test cases are classified as failing or passing, *CFAAR* operates as follows:

Step 1. It identifies a set of failure-causing control dependence chains that are minimal in terms of number and length. This is achieved by using an improvement of an existing *CBFL* technique [8], as described in Section 2.3.

Step 2. Using a heuristic search, it identifies a set of predicates within the chains along with associated execution instances, such that negating the predicates at the given instances would make some (but not necessarily all) of the failing tests exhibit correct behavior.

Step 3. For each candidate predicate, a classifier is built whose purpose is to dictate

when the predicate should be negated to yield correct behavior. The training output data for the classifier (to negate vs. not to negate) is deduced from the execution instances identified in Step 2. The training input data is derived from the program state captured at the point of predicate execution. It is worth pointing out that, in several cases we encountered, if Step 2 was only able to make part of the failing tests exhibit correct behavior, the built classifier might compensate for that shortcoming, as discussed in Section 2.2.3.

Step 4. For each classifier, *CFAAR* leverages a Decision Tree in order to synthesize a corresponding patch deployable in the form of a conditional statement guarding the candidate predicate. It should be noted that the user has the option to discard the synthesized patch and instead rely on the classifier to enable correct behavior at run-time; this might be a sensible option if the patch was deemed unmaintainable.

In this work, we make the following contributions:

1. An effective program repair approach that is centered on selectively altering control flow, with specific focus on defects that are repairable by negating control statements under specific conditions.

2. A *CBFL* component that identifies failure-causing control dependence chains.

3. A supporting toolset that targets the Java platform.

4. An evaluation of the toolset demonstrating its effectiveness at generating synthesized patches for the Introclass [9] benchmark and part of the Siemens benchmark [10].

5. Due to Java's Class Reloading capability, the synthesized patch or the corrective classifier could be dynamically deployed without requiring the application to be restarted

6. The current focus of *CFAAR* is seemingly narrow; however, we believe that the essence of the approach is extendable to address defects that are repairable by a variety of alterations to a program's control flow and even data flow.

1.2 D-FUSE: A Failure Detection Technique

When compared to Automated Program Repair, failure detection is considered a relatively conservative technique that can be used to produce a defensive fix. After detecting that a bug is executed, a system can be configured to revert the current transaction, or send a critical warning to the development and operational teams. While both APR and Failure Detection techniques can use common features and tools, the end resolution is different. One attempts to fix the problem at the application level by producing a correct output, while the other relies on other parts of the system and reports the failure.

A baseline failure detection technique can be built using simple *structural* profile elements. For instance, if an *if-statement* branch is directly associated with a vulnerability, the program run can be declared as failing when the branch is visited at runtime. By analyzing *structural* profiles for failing and passing runs, determining the branch associated with the vulnerability is straightforward. Also, by building custom instrumenters and profilers, it is also straightforward to generate a modified executable that raises a signal when the suspicious branch is executed. More sophisticated approaches such as [11] or [12] can make use

of a combination of *structural* profile elements to detect if the vulnerability was executed.

However, *structural* profiles are not always sufficient in identifying when a defect/vulnerability is executed. The contributing factors to a failure can lie in the state of the program. Unlike *strucural* profiles, which usually have a limited number of elements, state information can be an unlimited number of possibilities. However, *substate profiling* is a recent technique that reduces the unlimited vector space of the program state to a limited subspace. It was shown to be effective in Test Suite Reduction [13] and Fault Localization [14]. In this dissertation, we propose Detecting Failure Using Substate Profile Elements (*D-FUSE*), a novel approach that enables Failrue Detection using both *structural* and *substate* profiles. The approach works as follows:

Step 1. We first generate *structural* and *substate* profiles for the *subject program* for all test cases. In the case of *substate* profiles, the profiles are clustered into distinct profile elements.

Step 2. The resulting profiles for each test case along with its failure information are used in an Integer Programming (IP) optimization engine. The target of the optimization is to select a combination of elements that predicts failure, while minimizing the profiling cost. Multiple weighing techniques can be used to perform this minimization.

Step 3. By customizing the *structural* and *substate* instrumenters, we generate a minimally augmented set of probes to accurately detect when the profile element is satisfied.

(a) For structural profiles, compound profile elements that depend on a sequence of events A and B require that no event C can invalidate

the sequence. For example, to validate that a variable is defined at location $\mathfrak{l}_1$ and used in location $\mathfrak{l}_2$, we need to validate that the variable is not redefined at a third location $\mathfrak{l}_3$. This would increase the number of probes. On the other hand, combining multiple profile elements can make use of common probes, thus reducing the set of probes. Our aim is to use a minimal set of probes without sacrificing accuracy.

(b) For *substate* profiles, we augment the descriptor with the cluster information to accurately detect if the run belongs to the appropriate cluster.

Step 4. As a final step, a guard code is added to the subject program to apply the rules generated in Step 2. This code throws an error if a failure is imminent.

We tested our proposed method on the *Defects4J* framework [15] and showed the complementary nature of *structural* and *substate* profiles for failure detection. The results show that the proposed approach detects failures with an accuracy superior to the baseline approach. The methodology and results are further detailed in Chapter 3.

1.3 Dissertation Outline

The rest of the document is organized as follows. Chapter 2 presents our APR approach and its results. In Chapter 3, we illustrate our failure detection approach. Chapter 4 summarizes the related literature, and Chapter 5 concludes.

Chapter 2

Control Flow Alteration to Assist Repair

In this chapter, we present *CFAAR*, a program repair assistance technique that operates by selectively altering the outcome of suspicious predicates in order to yield expected behavior. *CFAAR* is applicable to defects that are repairable by negating predicates under specific conditions.

CFAAR proceeds as follows: 1) It identifies predicates such that negating them at given instances would make the failing tests exhibit correct behavior. 2) For each candidate predicate, it uses the program's state information to build a classifier that dictates when the predicate should be negated. 3) For each classifier, it leverages a *Decision Tree* to synthesize a patch to be presented to the developer.

We evaluated our toolset using 149 defects from the *IntroClass* and *Siemens* benchmarks. *CFAAR* identified 91 potential candidate defects and generated plausible patches for 41 of them. Twelve of the patches are believed to be correct, whereas the rest provide repair assistance to the developer.

The main contributions highlighted in this chapter are:

1. A program repair assistance approach that is centered on selectively altering control flow, with specific focus on defects that are repairable by negating control statements under specific conditions.

2. A supporting toolset that targets the Java platform.

3. An evaluation of the toolset demonstrating its effectiveness at generating synthesized patches for the *Introclass* benchmark and part of the *Siemens* benchmark.

Section 2.1 provides an introductory overview to the *CFAAR* program repair assistance technique. The detailed description of *CFAAR* is then provided in section 2.2 and in [5]. Section 2.3 describes the *CBFL* component while focusing on the proposed modifications. Section 2.4 presents the evaluation of *CFAAR* when applied to the *Introclass* and *Siemens* benchmarks.

2.1 Overview

Once a failure is detected, it is typically handed over to the developers in order to initiate the debugging process that involves: 1) identifying its root cause, and 2) modifying the code to prevent it from recurring. Researchers working on automating the debugging process refer to the first activity as fault localization, and the second as program repair. For over three decades, researchers have proposed a plethora of automated fault localization techniques and tools [16, 17, 18, 19, 20, 21, 22, 23, 24]. And in recent years, a number of automated program repair techniques have been proposed that leverage varying approaches such as evolutionary algorithms [9, 25], constraint solving [26, 27, 28, 29, 30],

and program mutation [31]. Long and Rinard [32], Xiong et al. [33], Demarco et al. [26], and Xuan et al. [34] proposed repair techniques that are focused on condition synthesis, which pertains most to our work.

We present *CFAAR* (*Control Flow Alteration to Assist Repair*), a test-based program repair assistance technique that operates by selectively altering the outcome of suspicious predicates in order to yield expected behavior, and subsequently provide a synthesized patch. It focuses on the category of defects that are repairable by negating control statements under some specific conditions. Unlike most other test-based repair techniques that mine for patches in other parts of the program [6, 35] or in various artifacts, *CFAAR* relies on the program's state to determine when a candidate control statement should be negated in order to yield correct behavior. The captured state information is further analyzed in order to synthesize a patch in the form of a conditional that guards the candidate control statement. When presented with a patch, the developer would: 1) use it as is, if deemed correct; or 2) use it as assistance during the debugging process.

Specifically, given a test suite in which the test cases are classified as failing or passing, *CFAAR* operates as follows:

Step1. It identifies a set of suspicious predicates using an existing coverage-based fault localization (*CBFL*) technique.

Step2. For each suspicious predicate, it uses a heuristic search to identify execution instances such that negating the predicates at the given instances would make some (but not necessarily all) of the failing tests exhibit correct behavior. Our repair assistance approach would be deemed to have failed in case this step was unable to make any failing test case exhibit correct behavior.

Step3. For each candidate predicate, a classifier is built whose purpose is to dictate when the predicate should be negated to yield correct behavior. The training output data for the classifier (to negate vs. not to negate) is deduced from the execution instances identified in the previous steps.

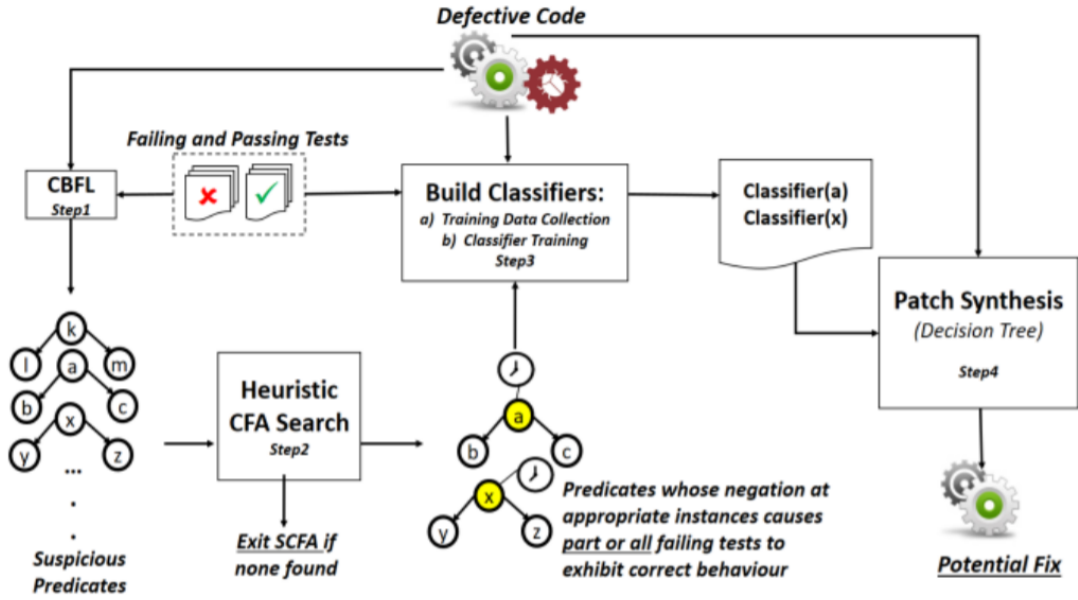
Step4. The training input data is derived from the program state captured at the point of predicate execution. It is worth pointing out that, in several cases we encountered, if *Step 2* was only able to make part of the failing tests exhibit correct behavior, the built classifier might compensate for that shortcoming, as discussed in Section 2.2.

Step5. For each classifier, *CFAAR* leverages a *Decision Tree* in order to synthesize a corresponding patch deployable in the form of a conditional statement guarding the candidate predicate. The developer might deem the patch correct and adopt it as a fix, or simply use it to guide and assist the debugging process.

2.2 CFAAR: Selective Control Flow Alteration

Our program repair approach is based on the premise that a measurable proportion of the defects are likely to trigger erroneous branch executions within highly suspicious control dependence chains. As such, we expect that properly altering such chains at runtime is likely to cause the failure to disappear, and thus enable us to synthesize a permanent code fix. This section describes *CFAAR* in detail.

Figure 2.1: CFAAR Overview



2.2.1 Overview

Figure 2.1 provides an overview of our approach. Given a set of passing test cases and another of failing test cases, the *CBFL* technique described in Section 2.3 is applied. The outcome is a set of highly suspicious control dependence chains that are minimal in terms of number and length.

A ***Heuristic Search for Control Flow Alterations*** (see Section 2.2.2) is applied on the suspicious chains to identify a minimal set of predicates whose negation at proper instances causes the failure to disappear in some or all test cases that were originally failing. *CFAAR* is deemed unsuccessful in case the search failed to find any candidate predicates and associated execution instances.

The ***Build Classifiers*** phase creates for each candidate predicate p a classifier whose purpose is to dictate when the predicate should be negated to yield correct behavior. This involves two steps:

1. *Training Data Collection*: the test suite is executed in order to capture the

program states relevant to p, every time p is executed. The captured states are labeled as those for when p needs to be negated and those for when a negation is not required.

2. *Classifier Training*: involves using the collected states to train a Decision Tree classifier that decides whether or not to negate p.

The ***Patch Synthesis*** phase generates a code fix by: a) building a *Decision Tree* out of the classifier and b) converting the tree into a predicate that will guarantee that the corresponding suspicious candidate predicate is negated appropriately. That is, the synthesized patch should faithfully replace the classifier.

2.2.2 Heuristic CFA Search

We devised ***HeuristicCFASearch***, a search algorithm that identifies which predicates to negate and when to negate them. Specifically, the goal is to identify according to which pattern of execution, picked from the list of pre-determined patterns shown in Figure 2.2, a certain predicate should be negated. For example, the "all" pattern means that the predicate should be negated all the time, and the "first+last" patterns means that it should be negated only the first time it executes and the last time.

Note how each of the supported patterns of execution are generic enough to be matched across different test runs. Consider for example a predicate p that executes 7 times in failing test case $\mathtt{t}_1$ and 8 times in $\mathtt{t}_2$. Now assume that we discover that $\mathtt{t}_1$ passes if p is negated according to the pattern "0111110". Finding a matching pattern in $\mathtt{t}_2$ is possible and yields "01111110" according to the generic pattern "all-(first+last)". However, if the discovered pattern was "1101011" it is not possible to find a unique match in $\mathtt{t}_2$. For that reason, we restrict our search

Figure 2.2: Heuristic CFA Search Algorithm

```
HeuristicCFASearch(PredList_susp, T_fail)

1. Patterns = {"all", "first", "last", "all-first", "all-last",
   "all-(first+last)", "first+1", "last-1", "first+last", "odd", "even"}

2. PredList_solution = Ø

3. for each p in PredList_susp  do
4.   for each t_fail in T_fail do
5.     for each pattern in Patterns do
6.       execute t_fail while negating p according to pattern
7.        if execution succeeds
8.          p.repairs(t_fail, pattern)
9.          PredList_solution = PredList_solution U p
         endif
       endfor
    endfor
endfor

// order the (p, pattern) pairs by the number of the test cases they fix

PredPatternPQ_solution = Ø              // priority queue

10. for each p in PredList_solution do
11.   for each pattern in Patterns do
12.        priority = p.getNumFixedTCsByPattern(pattern)
13.        PredPatternPQ_solution.insert(priority, (p, pattern))
    endfor
endfor

 14. return PredPatternPQ_solution
```

to patterns that are easily reproducible across different execution runs.

HeuristicCFASearch considers a single suspicious predicate p at a time and a set of different patterns of negation. Specifically, it checks whether any of the following actions would make some or all of the failing test cases succeed: 1) negating p all the time within a given failing test case; 2) negating p the first time; 3) negating p the last time; 4) negating p all the time except the first; 5) negating p all the time except the last; and so on, as indicated on Line 1 of the pseudo-code shown in Figure 2.2.

HeuristicCFASearch takes as input: 1) $\mathbf{PredList}_{\mathtt{susp}}$: the list of suspicious predicates identified by the *CBFL* component; and 2) $\mathsf{T}_{\mathtt{fail}}$: the set of failing test cases within the training set. Line 1 initializes *Patterns* with the execution patterns to be matched. Note that the patterns are roughly ordered in terms of their simplicity. On Line 2, $\mathbf{PredList}_{\mathtt{solution}}$ is initialized to the empty set. Its role is to store the suspicious predicates that are candidates for repairing one or more failing runs. For every suspicious predicate p, every failing test $\mathtt{t}_{\mathtt{fail}}$, and every pattern *pattern*, Line 6 executes $\mathtt{t}_{\mathtt{fail}}$ while negating p according to *pattern*. In case the execution succeeds, p is deemed to be a viable candidate for repairing $\mathtt{t}_{\mathtt{fail}}$ according to *pattern*. Accordingly, Line 8 associates p with $\mathtt{t}_{\mathtt{fail}}$ and *pattern*, and Line 9 adds p to $\mathbf{PredList}_{\mathtt{solution}}$.

Lines 10-13 order all of the *(p, pattern)* pairs based on the number of failing test cases they fixed. The ordered list is stored in the priority queue $\mathbf{PredPatternPQ}_{\mathtt{solution}}$, and returned at Line 14.

2.2.3 Building the Classifiers

Training Data Collection - We train one classifier at a time for every pair *(p, pattern)* in the ordered list identified by ***HeuristicCFASearch***. The objective

is to obtain one or more classifiers that can *plausibly* fix the subject program. If multiple classifiers turn out to be plausible, the one that facilitates synthesis the most will be considered.

Given a pair *(p, pattern)* we collect data to train a classifier that will guide the execution by indicating when to negate p according to *pattern*. Two sets of data are actually needed, one associated with when p needs to be negated and another associated with when p should remain intact. In other words, we need to capture the states induced by: 1) the failing test cases that were fixed using *(p, pattern)*; and 2) all the passing test cases.

The two sets are built by collecting the approximated state of the program right before each execution of p. Specifically, on the onset of p executing, the values of the following entities are collected:

1. *Use(p)*, i.e., the local variables, static variables, array elements, instance fields, and method return values, directly *used* in p.

2. Formal parameters of the method containing p.

3. Local and static variables that were *used* or *defined* within the method containing p.

The values of the program variables are derived according to their types, as follows:

1. Variables of type *float* and *double* have their values used as is, i.e., as scalars.

2. Variables of type *int*, *long*, *char*, *byte*, and *short* have their values used in a dual manner, as scalars and categorical.

3. *java.lang.String* objects are categorically represented within the classifier, such that the categories are determined based on the *java.lang.String.hashCode()* method.

4. Non-*String* objects are also categorically represented within the classifier. However, the categories are determined based on a hash code computed by considering the states of the objects' attributes, and if need be, by recursively considering the attributes of their attributes and so on. In other words, to represent a non-*String* object, a hash code is first derived based on all of its direct and indirect attributes.

Training the Classifiers - In this phase, a classifier is trained using the previously collected training data. The outcome is a *Decision Tree* that tests one variable at a time to determine if the candidate predicate should be negated. Since the operations that process decision trees are greedy by nature, we expect to have a small number of variables in the tree and consequently only few variables in the resulting synthesized patch.

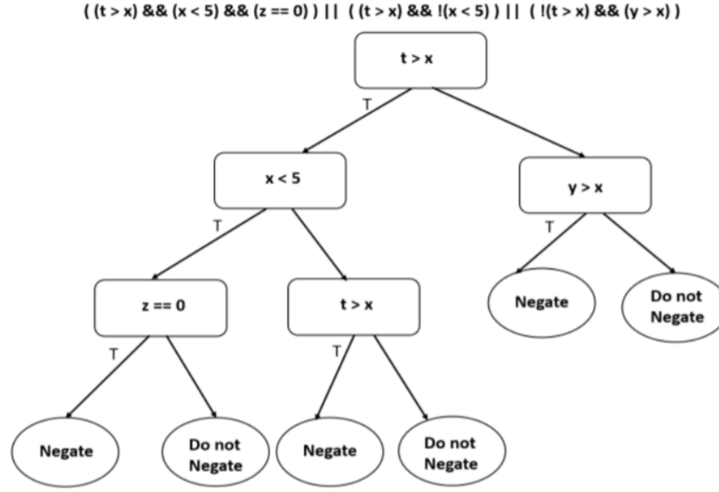
It is worth noting that during our work we encountered an unexpected but serendipitous scenario related to classifier training. The following illustrates it:

1. ***HeuristicCFASearch*** was only successful at making part but not all the failing test cases pass.

2. As a result of training the classifiers, one of the classifiers captured the invariant "$x > 5$ when predicate p must be negated".

3. When rerunning the program and negating p whenever $x > 5$; unexpectedly, all (as opposed to part) of the test cases exhibited correct behavior. That is, the captured invariant generalized to all failing test cases without affecting

Figure 2.3: Decision Tree Conversion Example



the passing test cases. In our experiments, this scenario occurred several times.

Patch Synthesis

Given a classifier that makes all test cases pass, the ***Patch Synthesis*** phase generates a synthesized patch by building a *Decision Tree* out of the classifier, and then converting it into a predicate that guards the suspicious candidate predicate. This process is detailed below and illustrated in Figure 2.3.

The computed decision tree serves as a blueprint of the patch. Its leaves indicate whether the predicate should be negated or not. The tree is first converted to a Boolean expression as follows (see Figure 2.3):

1. Every path from the root node to a leaf indicating a negation is transformed into a rule comprising a conjunction of the conditions along the path.

2. The obtained rules are grouped using disjunctions.

3. The final expression is a DNF formula consisting of literal equalities and

Figure 2.4: Instrumentation for Negating Predicates

// Original Code
if (x<0) // identified as 1001
    x=x+1;
else
    y=y+1;
z=z+1;
		// Instrumented Code
		if ((x<0) ^ shouldNegate(1001))
		    x=x+1;
		else
		    y=y+1;
		z=z+1;	

// Original Byte Code		**// Instrumented Byte Code**	
7 iload_1 8 ifge 14 9 iload_1 10 iconst_1 11 iadd 12 istore_1 13 goto 18	14 iload_2 15 iconst_1 16 iadd 17 istore_2 18 iload_3 ...	7 iload_1 **8 sipush 1001** **9 invokestatic** **10 shouldNegate(I)Z** **11 ifeq 14** **12 ifge 15** **13 goto 20** **14 ifge 20** 15 iload_1 16 iconst_1	17 iadd 18 istore_1 19 goto 24 20 iload_2 21 iconst_1 22 iadd 23 istore_2 24 iload_3 ...

inequalities. We note that the generated expression can be reduced through various methods, but this doesn't affect the correctness of the expression.

The resulting Boolean expression is transformed into bytecode following the steps below:

1. The variables used in the patch are identified. Local variables and static fields can be used directly in the patch. However, instance fields and method returns cannot; they are stored in temporary local variables that the patch will use.

2. Compatible bytecode operators for the equalities and inequalities in the Boolean expression are identified. For example, a variable of type int will require a different comparison operator than what a variable of type *double* would require.

3. The Boolean expression is transformed into a sequence of *if* statements that determine if the condition is met or not.

4. A guard is created just before the candidate predicate. It executes an alternative *if* bytecode with flipped targets when the condition is met. The guard is identical to the one presented in Figure 2.4.

Implementation

Our implementation targeted the Java platform at the byte code level. Part of the work that posed most implementation challenges included *Heuristic CFA Search*, and *Training Data Collection* that both involved instrumenting and profiling Java byte code using the Byte Code Engineering Library, BCEL.

Training Data Collection calls for developing a state profiling engine that captures a snapshot of the approximated program state at given code locations, as described in Section 2.2.3. The profiling engine consists of two main subcomponents: the *Instrumenter* and the *Profiler*. The preliminary step is to instrument the target byte code class files using the *Instrumenter* which inserts a number of method calls to the *Profiler* at given points of interest. At runtime, the instrumented application invokes the *Profiler*, passing it information that enables it to log the approximated program states.

Heuristic CFA Search aims at identifying which predicates to negate and following which pattern. In order to enforce a given pattern, the number of occurrences of each suspicious predicate needs to be known a priori, thus requiring each failing test to be executed twice. To support the conditional negation of a given predicate (Line 6 of *HeuristicCFASearch*), each suspicious predicate is augmented by byte code that enables the *CFA Search* profiler to negate the predicate when needed.

Figure 2.4 is an illustration of how a predicate would be instrumented. The ID of the predicate is pushed on the stack (Line 8) and a method *shouldNegate()* is invoked (Lines 9-10) right before the if statement executes. The returned value is checked (Line 11). If the returned value is false, the code branches to the original if statement (Line 14). If the returned value is true, a synthesized if statement with swapped branches is executed (Lines 12-13). On the profiler side, *shouldNegate()* logs the timestamp and the ID of the executed *if* statement, and returns true or false according to *HeuristicCFASearch*.

2.3 CBFL: Identifying Failure-Causing Predicates

CFAAR requires a very small number of highly suspicious predicates to be first identified. Using a traditional coverage-based fault localization technique that uses simple profiling elements (e.g. branches) might not always be effective since the defect might be too complex to be characterized by any set of elements; thus, no suspicious predicates would be identified. Also, using structural profiling elements that are more complex than the defect at hand might identify too many suspicious predicates. In both cases the *CFAAR* requirement would not be satisfied.

A *CBFL* technique most likely to satisfy the *CFAAR* requirement is one that uses profiling elements that closely characterize the defect at hand, i.e., elements that are structurally no less nor more complex than the defect. The *CBFL* technique presented in [8] seems most suitable. It uses program elements that vary in complexity in order to better match the complexity of the defect of concern, namely, dependence chains with varying lengths. In summary, the technique operates as follows: 1) it starts by using the simplest profile elements, i.e., de-

pendence chains of length one; 2) if one or more failure-correlated elements are found, it stops; 3) otherwise it uses the next more complex profile elements, i.e., dependence chains of length two; and so on until a failure-correlated chain is found or the resources are exhausted.

In order to make the above technique [8] suitable for *CFAAR*, we modified it and extended it as follows:

1. We used control dependence chains only as opposed to data and control dependence chains.

2. We improved its accuracy by considering the causal relationships amongst program statements; thus enabling it to identify failure-causing as opposed to failure-correlated predicates, as described next.

2.3.1 Causal Inference: Background

Causality [36, 37, 38] is clearly more desirable than correlation for the purpose of fault localization, since the ultimate goal is to identify and repair the code that caused the failure and not just any code that correlated with it. Early *CBFL* technique erroneously used correlation to compute the suspiciousness score in order to infer the causal effect of individual program elements on the occurrence of failure. The scores they used suffer from confounding bias, which occurs when an apparent causal effect of an event on a failure may actually be due to an unknown confounding variable, which causes both the event and the failure. Confounding bias might explain the high rate of false positives exhibited by such techniques [39].

Given a program and a test suite, assume for example that all failing test cases induce dependence chain $e1 \rightarrow e2 \rightarrow e_{\text{bug}} \rightarrow e3 \rightarrow e4 \rightarrow e_{fail}$ and all

passing test cases induce $e1 \rightarrow e2$ only; where e_{bug} exercises the fault and $e_f ail$ indicates a failure. A correlation-based approach would determine that any of e_{bug}, e_3, or e_4 is equally suspect to be the cause of the failure, thus resulting in two false positives. Whereas, a causation-based approach that considers dependencies to have causal effect, would determine that e_4 is the least suspect and e_{bug} the most suspect. This is because: 1) confounding bias weakens the causal relationship; and 2) when computing the suspiciousness scores, the confounding bias to consider for e_4 would involve e_3 and e_{bug}, for e_3 it would involve e_{bug}, and no confounding is involved when computing the suspiciousness score of e_{bug}.

Confounding bias is a common phenomenon that needs to be identified, controlled, and reduced. Baah et al. were the first to investigate the application of causal inference in *CBFL* [40] [41].

Given a statement s in program P, the aim of their work is to obtain a causal-effect estimate of s on the outcome of P that is not subject to severe confounding bias, i.e., a causation-based suspiciousness score of s. They applied Pearl's Back-Door Criterion [36, 37] to program control dependence graphs in order to devise an estimator based on the following linear regression model:

$Y_s = \alpha_s + \tau_s T_s + \beta_s C_s + \epsilon_s$

This model relates the event of program failure Y_s with not just the event of covering statement s (i.e., T_s), but also with the confounding events, listed in C_s. The model is fit separately for each statement s, using statement execution profiles that are labelled as passing or failing.

This work assumes that: 1) if s is faulty, covering it will cause a failure; and 2) if s is dynamically directly control dependent on statement s', s' causes the execution of s and possibly the failure, i.e., s' is the only source of confounding bias and C_s becomes a single indicator of whether s' was covered. More impor-

tantly, since τ_s is the average effect of T_s on Y_s, the work uses an estimate of τ_s to quantify the failure-causing effect of s, i.e., the suspiciousness of s.

Note that neither of the above assumptions is sure to hold. For example, due to coincidental correctness, covering a faulty statement will not necessarily cause a failure; and due to the transitivity of control/data dependencies in programs, direct control dependencies might not be the only source of confounding bias. Nevertheless, using the above model is likely to yield more accurate suspiciousness scores than simply relying on correlation.

2.3.2 From Failure-Correlated to Failure-Causing

Refining the failure-correlated chains into failure-causing chains is an important step to arrive at a more accurate fault localization that eventually leads to a more effective program repair. To do so, we first compute the causal effect for each statement s appearing in the failure-correlated chains using the approach adopted in [40] as follows:

1. Fit a linear regression model in the form of $Y_s = \alpha_s + \tau_s T_s + \beta_s Cs + \epsilon_s$ as discussed in Section 2.3.1. In our case, C_s represents the statement upon which s is directly control-dependent. That is, C_s would be 1 if a given test cases covers the control predecessor of s and 0 otherwise. If s has no control predecessor, the model to fit would be $Y_s = \alpha_s + \tau_s T_s + \epsilon_s$.

2. The causal effect of s is estimated via the coefficient τ_s. To refine the failure-correlated chains, we sort them based on the maximum causal effect per chain and select the top three.

Table 2.1: Information about Subject Programs

		Defects	—T—	—Tlarge—	LOC
Siemens	*print_tokens*	4	4070	-	536
	print_tokens2	8	4055	-	387
	replace	6	2843	-	554
	schedule	4	2650	-	425
	tot_info	17	1052	-	494
	tcas	18	1597	-	136
IntroClass	*digits*	20	16	1000	15
	grade	20	18	1000	19
	median	20	13	1000	24
	smallest	20	16	1000	20
	syllables	4	16	1000	23
	checksum	7	16	1000	13

2.4 Empirical Results

As our work in [5], this section tries to answer the following research questions:

1. **RQ1**: How Prevalent are the Defects that are Potentially Repairable by CFAAR?

2. **RQ2**: How Effective is CFAAR at Synthesizing Plausible Patches?

3. **RQ3**: How Effective is CFAAR at Synthesizing Correct Patches?

In order to address these questions we applied our toolset to 12 Java programs for a total of 148 defects. Next, we describe the used subject programs then present and discuss our results.

2.5 Subject Programs and Test Suites

Our experiments involved 57 defective versions from the *Siemens* benchmark (*sir.unl.edu*) and 91 versions from the Introclass benchmark [42]. The *Siemens*

subjects, namely, 8 *print_tokens2* versions, 4 *print_tokens* versions, 6 *replace* versions, 4 *schedule* versions, 1 *schedule2* version, 18 *tcas* versions, and 17 *tot_info* versions were manually converted to Java in [8]. Note that we excluded irrelevant bugs, those that could not be converted from C to Java, or those whose Java versions did not fail or caused exceptions to be thrown. The Introclass benchmark is originally written in C, it contains 6 programs (*digits*, *grade*, *median*, *smallest*, *syllables*, and *checksum*) and hundreds of related bugs. We opted to randomly select 20 versions from each program and convert them to Java. As a result, we used 20 *digits* versions, 20 *grade* versions, 20 *median* versions, 20 *smallest* versions, 4 *syllables* versions, and 7 *checksum* versions, for a total of 91 versions.All subject programs are downloadable from [43].

Table 2.1 summarizes the information regarding the defective versions we used in addition to the test suite sizes. Note that the original test suites for the *Introclass* programs are very small; therefore, we randomly generated an additional larger test suite for each, referred to as T_{large} in Table 2.1. However, some versions did not fail using T_{large}, for those we used the original smaller test suite, denoted by T in the table.

2.6 RQ1: How Prevalent are the Defects that are Potentially Repairable by CFAAR?

A defect that is potentially repairable by *CFAAR* is one that could be fixed by negating one of its predicate statements at some instances during execution. In the context of our work, in order to get an estimated answer, we will assume that it is any defect for which ***HeuristicCFASearch*** makes at least one failing test exhibit correct behavior. Clearly, this is not a very accurate estimate since (cur-

Table 2.2: Summary of Results

	Siemens (T)	IntroClass (T & Tlarge combined)
#Versions	57	91
Partially Fixed by HeuristicCFASearch	32	59
Fully Fixed by HeuristicCFASearch	20	25
Partially Fixed by Classifiers	31	59
Plausible Patches	20	17
Correct Patches (checked manually)	3 (at least)	-
Correct Patches (checked via testing)	-	4

rently) ***HeuristicCFASearch*** only explores a limited number of patterns, and only considers one predicate at a time as opposed to combinations of predicates.

The second row in Table 2.2 shows for each benchmark the number of the versions for which ***HeuristicCFASearch*** made some or all failing test cases behave correctly.

The third row shows for each benchmark the number of the versions for which ***HeuristicCFASearch*** made some or all failing test cases behave correctly.

The fourth row shows the numbers for which ***HeuristicCFASearch*** made all failing test cases behave correctly. On average, 58% of the versions had some or all of their failing test cases pass, and 30% had all of them pass. These findings suggest that the applicability of *CFAAR* is not very narrow.

These findings suggest that the current focus of *CFAAR* is not as narrow as we initially thought it would be.

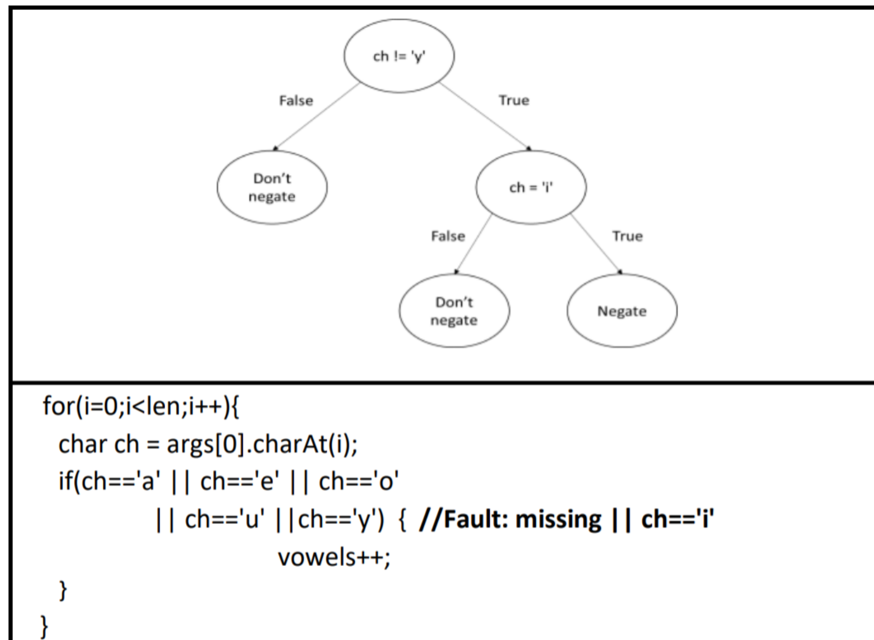
2.7 RQ2: How Effective is CFAAR at Synthesizing Plausible Patches?

Whenever a given classifier was successful at making all failing test cases behave correctly, *CFAAR* will synthesize a corresponding plausible patch. Recall that a *plausible patch* is one that makes all the test cases pass (including those that were failing before the patch). Note that in many cases, multiple plausible patches could be generated for each defect, which calls for ranking them *w.r.t* likelihood of correctness.

The fifth row in Table 2.2 shows for each benchmark the number of the versions for which the classifiers fixed some or all of the failing test cases. The sixth row shows the numbers for which the classifiers fixed all of the failing test cases. The numbers shown in the sixth row also represent the number of plausible patches synthesized by *CFAAR*. Therefore, *CFAAR* was successful at generating plausible patches for 41 out of the 149 defects (i.e., 27.5%).

We now illustrate *Patch Synthesis* using a correct patch for one subject version used in our study, namely, *syllables v1*, *grade v13*, and *tcas v1*.

Example 1 - The original code for *syllables v1* is faulty as it fails to check whether *ch* is equal to *i*:



The above figure shows a decision tree associated with one of the plausible patches for *syllables v1*.

The synthesized patch suggests replacing ch==‘y’ with:

```
ch=='y' ^ ((ch!='y' && ch=='i'))    // where ^ is xor
```

This plausible patch happened to be correct as it can be shown that it is semantically equivalent to the real fix:

```
ch=='y' ^ ((ch!='y' && ch=='i'))
⇔  ch=='y' ^ ch=='i'
⇔ (ch!='i' && ch=='y' || ch=='i' && ch!='y')
⇔ (ch='y' || ch='i')
```

Example 2 - The original buggy code for *grade v13* is faulty because it mistakenly checks if the score is greater than a and b, instead of between a and b, as shown next:

```
if (score >= a){
    result += 'A';
}
else if ((score >= b) && (score > a)){ /* Fault –
            Potential fixes:  1) remove (score > a)
                              2) replace with (score < a) or (score <= a) */
    result += 'B';
}
else if ((score >= c) && (score < b)){
 ..
```

The decision tree consists of a single "negate" leaf node to be applied on the clause ($\mathtt{score} > \mathtt{a}$). Therefore, the synthesized patch suggests replacing ($\mathtt{score} > \mathtt{a}$) with !($\mathtt{score} > \mathtt{a}$) or ($\mathtt{score} <= \mathtt{a}$).

Example 3 - The previous two example patches happened to be correct. Applying *CFAAR* on *tcas v1* below yields a plausible patch that is actually incorrect:

```
result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
(!(Down_Separation > ALIM()))); //fault: should have >=ALIM
```

Given the test suite associated with *tcas*, negating the faulty condition all the time was enough to make all test cases pass, which is clearly not a correct fix.

To improve the quality of our fixes, complementary approaches can be considered:

1. Improving the test suite by having more test cases cover the suspicious condition to help fine-tune the generated patches.

2. Ranking and prioritizing patches by looking at features such as syntactic/semantic distance to the faulty code, and similarity with documentation [33] and previous fixes [44].

Table 2.3: Comparison to state-of-the-art results

	ACS	JGenProg	Nopol	CFAAR
Defects	224	224	224	149
Plausible	23	24	35	41
Correct	18	2	5	12
Incorrect	5	22	30	29
Precision	78.3%	18.5%	14.3%	29.3%
Recall	8.0%	2.2%	2.2%	8%

2.8 RQ3: How Effective is CFAAR at Synthesizing Correct Patches?

In order to assess our confidence in the correctness of the patches synthesized by *CFAAR* we follow two approaches. In case of *IntroClass*, we tested the patched subjects using validation test suites that we generated. The validation tests were programmatically created (rather easily) by generating random inputs. Out of the 17 plausible patches in *IntroClass*, 13 failed. That is, we have high confidence that 4 of our patches are correct.

Concerning the 20 *Siemens* plausible patches, using validation test suites was not feasible since it is hard to generate additional tests for these programs (noting that we used all existing tests for training). In this case, we opted to select a subset of the plausible patches to examine manually. The subset included 7 patches, of which we believe that 3 are correct and 4 are incorrect.

In the Appendix, we show 8 of the correct program patches. Table 2.3 compares our results to reported ACS, JGenProg, and Nopol results in [33]. In this comparison, we consider the number of versions that are potentially fixable by CFAAR as the total number of defects, which is 91.

2.9 Threats to Validity

A major threat to the external validity of our approach is the fact that our experiments involved a limited number of subject programs and faults; therefore, it is not possible to draw broad conclusions based on our results. This could be remedied by conducting further experiments involving a variety of other subject programs from different domains and environments containing real and/or seeded defects.

We recognize the following threats to the internal validity of our approach:

1. Our *CBFL* approach assumes that most defects could be characterized by a few control dependence chains of some limited length. Actually, it is plausible that some defects might not be characterized by any structural profiling elements no matter how complex they are.

2. In its current state, *CFAAR* targets a rather narrow category of defects. However, we believe that the basic approach behind *CFAAR* is extendable to address defects that are repairable by a variety of alterations to a program's control flow and even data flow; which we intend to address in future work. Here we are referring to the methods adopted by *CFAAR*: a) to heuristically search for instances for when an alteration should be applied; b) to build classifiers based on state information; and c) to synthesize patches based on the classifiers.

As it is the case for most test-based fault localization and program repair techniques, the effectiveness of our technique is greatly dependent on the quality of the test suite. This applies to both, identifying the suspicious chains and to training the classifiers. One way to tackle the issue of test suite quality is to leverage automated test case generation.

2.10 Analysis

In this chapter, we presented *CFAAR*, a test-based repair technique that targets defects that are repairable by negating control statements under some specific conditions. *CFAAR* relies on the program's state to determine when a candidate control statement should be negated in order to yield correct behavior. A synthesized patch is generated based on the state information, in the form of a conditional that guards the candidate control statement. Our experiments involving 149 defects revealed the following: 1) 91 defects were found to be potentially repairable by *CFAAR*; 2) 41 plausible patches were generated by *CFAAR*; and 3) at least 12 plausible patches are believed to be correct.

Chapter 3

Detecting Failure Using Substate Profile Elements

The National Institute of Standards and Technology (NIST) estimates that the cost of fixing a bug in production ranges between 30 and 60 times more than an early detection and fix [2]. A portion of this cost can be attributed to unplanned overhead to the development lifecycle, and to potential reputation loss before a fix is produced. For reasons like these, tools that help produce faster code fixes can help mitigate these costs.

Such tools often rely on analyzing the execution profiles of a subject program through various runs. For instance, detecting which events are associated with a defect or vulnerability can rely on profiles that are *structural* in nature, which mainly include the statements and branches that were covered in the code. However, some program defects cannot be characterized by structural information alone. State information, such as the value of the program variables, can complement the former profiles.

Recent advances in Software Engineering [13, 14] explored *substate* profiling, a

technique that extracts compact information about the state of the program. This technique has shown improvements to Software Engineering research problems like Fault Localization, Test Suite Reduction and Test Suite Prioritization.

Failure detection can be a conservative technique that produces a temporary fix to a known vulnerability before it is properly debugged. After detecting that a failure has occurred, a system can be configured to cancel/revert the performed transaction, or to raise a critical ticket to the appropriate teams for resolution. Alternatively, for benign bugs, a warning can be produced instead.

In this chapter, we present our method, **D**etecting **F**ailure **U**sing **S**ubstate Profiling **E**lement (D-FUSE), which studies how *substate* profiling, a novel profiling technique, can complement *structural* profiling in characterizing a defect or vulnerability. After showing the complementary nature of *substate* profiles and *structural* profiles for intrusion detection, we develop a framework to analyze *subject programs* and generate intrusion detection rules that get triggered when an intrusion occurs. Furthermore, we evaluate this framework on *Defects4J* [15], a state-of-the-art collection of faulty software libraries. We show how *substate* profiles complement *structural* profiles in detecting failing runs. Moreover, we show that our developed framework successfully finds feasible rules that identify failures in *Defects4J* subject programs. We also show that *structural* elements can predict 75% of failures, whereas *substate* profiles can predict 86% of failures, and using both element types can predict 93% of versions with defects. When *D-FUSE* was applied to these programs, intrusion detection failed in only 4-5% of the programs.

The rest of the chapter is structured as follows. We begin with some background on both *structural* and *substate* profiles in Section 3.1. We follow with a motivating example in Section 3.2. An empirical study is presented in Section

3.3 and the online failure detection approach is illustrated in detail in Section 3.4. Lastly, we provide a final analysis in Section 3.5.

3.1 Background

Profiling the faulty *subject programs* is a common preliminary step to many analysis techniques such as Fault Localization, Automated Program Repair, Test Suite Reduction, and Intrusion Detection Systems. The profiles are generated on an associated *Test Suite* containing passing and failing test cases for this program. Typically, the *subject program* is first run and profiled individually for each of these test cases, then, all profiles are used to optimize for the latter problems. In this section, we provide an overview of two types of profiling that are used in our study, namely, *structural* profiling and *substate* profiling.

3.1.1 Structural Profiles

Structural profiling is a common type of program profiling used in software Engineering analysis. It tracks information about program elements that are structural in nature, such as whether *basic-blocks* were executed in each run. More advanced profiling can track branching between pairs of *basic-blocks* that were executed consecutively, and pairs of statements containing where each variable is defined and used (i.e. *def-use pair*). These location pairs depend on the execution of the program, and can offer insights into the nature and location of the bug. Formally, we define the following terms.

1. *Basic-block*: is a code sequence with no branching. We denote by *BB* a feature type capturing if the basic block was visited at least once in a single run.

2. *Basic-block edge*: is the branching from one *basic-block* to another (e.g. entering an *if-statement* branch). We denote by *BBE* a feature type capturing if a *basic-block edge* was visited at least once in a run.

3. *Def-use pair*: is when a variable x is defined at statement s_1 and its value is used in statement s_2. We denote by *DUP* a feature type capturing if a *def-use pair* occurs at least once in a run.

Note that the number of possible *BB*, *BBE*, and *DUP* features can be determined statically from the subject program.

3.1.2 Substate Profiles

The tool first captures state information about three types of entities during a program run. Namely, these entities are: (1) variable definitions – including local variables, static variables, and fields, (2) return statements, and (3) formal parameters at function entries. For the purpose of our setup, a *capture point* (Cp) tracks the values of an entity at a unique program location. Note that a variable has as many Cp's as the number of times it was defined/assigned.

For a given test case, the tool tracks the values captured by each Cp. These values are then reduced to 17 statistical metrics (e.g. *mean*, *median*, and *isDescending*). Table 3.1 illustrates the collected features for a sample Cp after the entire test suite is profiled. In this example, a number of test cases n interact with this Cp, and each interaction can be characterized by 17 feature values. For each Cp, the test cases are then clustered using *k-means* [45] based on their feature values. We use a configuration similar to the one proposed in [14] to automatically determine the optimum number of clusters for each Cp.

Table 3.1: A sample of the features generated by one capture point

Test case Id	Number of Elements	Minimum	Maximum	Mean	Median	Standard Deviation	Inter-Quartile Range	Gini	Skewness	Median	Standard Deviation	Kurtosis	Mode	LongestZeroSequence	isIncreasing	isDecreasing	isConstant
23	f_1^{23}	f_2^{23}	f_3^{23}					...								f_{16}^{23}	f_{17}^{23}
38	f_1^{38}	f_2^{38}	f_3^{38}					...								f_{16}^{38}	f_{17}^{38}
...																	
85	f_1^{85}	f_2^{85}	f_3^{85}					...								f_{16}^{85}	f_{17}^{85}

Not all test cases visit this Cp. For each test case i, the features f_1^i - f_{17}^i are generated from the logged values. For every Cp, the test cases are clustered based on these feature values.

3.2 Motivating Example

We illustrate a sample program in the example in Table 3.2, showing how *structural* profiles do not capture any relevant information about the defect, while *substate* profiles do. A similar program is illustrated in a previous work [14]. The program converts an eight-digit binary number to an integer. However, an overflow occurs when the binary number starts with a 1.

A *structural* profile captures, for a given run, information related to *basic blocks* that were visited, and *def-use* pairs. However, note that all test cases execute all *basic blocks* and statements. Also, all *def-use* pairs are covered in every test case. For instance, the value of the variable *increment* at S9 is defined in both S6 and S8 for any given test case. Therefore, all *structural* profiles are similar and cannot discriminate between passing and failing tests.

On the other hand, *substate* profiles capture partial information about the

Table 3.2: Substate profiles and failure detection code

Given a string representing an 8-digit binary number, the method decimal() returns its decimal conversion. Due to overflow, failure occurs whenever the input string has its leftmost position set			Pass				Fail	
			t_1	t_2	t_3	t_4	t_5	t_6
public class BinarytoDecimal {			00101111	01011101	01111100	01111101	11101111	10110101
public static void main(String args[]) {	S1	Cp1.e1	x	x	x	x		x
		Cp1.e2					x	
decimal(args[0]); **validateRun();** }	S2							
public static int decimal(String binary) {	S3	Cp2.e3	x	x	x	x		x
		Cp2.e4					x	
int decimal = 0;	S4	Cp3						
for (int i = 0; i < binary.length(); i++) {	S5	Cp4 Cp5						
byte increment = 0;	S6	Cp6						
if (binary.charAt(i) == '1') {	S7							
increment = (byte) Math.pow(2.0 , (double)(7-i));	S8	Cp7.e5	x	x	x	x		
		Cp7.e6					x	x
updateInfo(S8, increment); }								
decimal += increment; }	S9	Cp8.e7	x	x	x	x		
	S9	Cp8.e8					x	x
return decimal; } }	S10	Cp9.e9	x				x	x
		Cp9.10		x	x	x		

The left side illustrates a faulty program. The top right side shows multiple test cases t_1-t_6 each comprising a binary input. Statement S8 is the faulty instruction that produces a wrong output for test cases t_5 and t_6. The observed values of each collection point (Cp) are clustered into one or more cluster (e.g. Cp_2 has two clusters *e3* and *e4*). Each test case is associated with (at most) one cluster per Cp. The lines in bold show the modifications needed to detect a program failure.

state of the variables in this program. Consider Cp_7, which captures the values of the local variable *increment*. For each test case, the values of *increment* were captured, and 17 features were calculated for Cp_7. Then, the test cases were clustered based on the feature values they were associated with. The same calculations were done for all Cp's.

Conveniently, one of the detected clusters ($\mathsf{Cp}_7.\mathsf{e}_6$) could directly predict failure. This can be attributed to features like *isDescending*, which becomes false only when an overflow occurs. Alternatively, $\mathsf{Cp}_8.\mathsf{e}_8$ is another cluster that is associated with failure.

To modify this program and enable it to detect when a fault occurs, we add can a call right after S8 to update the value of *increment*, and another call to validate the run right before the program exits. The program validation extracts the features from the values of Cp_7 and checks if they belong to cluster $\mathsf{Cp}_7.\mathsf{e}_6$. When they do, the program signals that the run is faulty.

3.3 Empirical Study

As a preliminary study, we generated *structural* and *substate* profiles for 222 *subject programs* exhibiting a single fault. These subject programs are based on 4 libraries in *Defects4J*, namely, *Chart*, *Time*, *Math*, and *Lang*. The setups are similar to the ones described in [13] and [14]. Structural profiles included *BB*, *BBE* and *DUP*. For *substate* profiles, 17 metrics were generated for each location point, which were later clustered into k clusters using *k-means* [45]. The value of k was determined dynamically using the elbow method.

To assess if profiles can characterize a bug, our goal was to find a single profile element that can discriminate between a failing or a passing test case.

Table 3.3: Number of versions having at least one profile element associated with failure

TOTAL # OF VERSIONS	COUNT C1>0	COUNT C2>0	COUNT C3>0	COUNT C4>0
222	157	122	185	143
100%	71%	55%	83%	64%

The profiling element needs to be covered only in a failing test case, and not in a passing test case.

For every subject program, we collected the following values:

- C1: the number of *structural* profile elements identifying at least one failing test case
- C2: the number of *structural* profile elements identifying all failing test cases
- C3: the number of *substate* profile elements identifying at least one failing test case
- C4: the number of *substate* profile elements identifying all failing test cases

The number of subject programs that have at least one discriminating profile element is reported in Table 3.3. For instance, 55% or 122 versions have at least one *structural* profiling element that is directly associated with failure, whereas 64% or 143 versions have a *substate* element that is directly associated with failure.

A more detailed analysis shows that the failure of 42 versions, which was not detectable by a single structural element, are detectable using a single *substate* element. And vice-versa, the failure of 21 versions is detectable with structural elements, but not *substate* elements. This finding confirms the theoretical com-

plementary nature of *substate* profiles to structural profiles, and in this case, their application to Intrusion Detection Systems.

3.4 Online Failure Detection

Our preliminary analysis in the previous section showed the complementary nature of *structural* and *substate* profiles. To enable the subject program to detect failures, we augment it in a way similar to the method described in Section 3.2 and Table 3.2. Namely, after determining which profile elements are correlated with failure, we insert probes at the corresponding `Cp` locations to track their values, then check which clusters these values match before the program exits in order to determine if a failure occurred. Similarly, structural profiles can be matched at runtime using a similar technique. We insert probes to track each structural element associated with failure. Finally, we check if these probes were triggered right before the program exits and send a failure signal, which would trigger an appropriate action in the production environment.

In addition to the setup above, several challenges should be considered:

- The cost of profiling in the final production code needs to be minimized.
- Failure can be associated with multiple elements. It can occur when two profiling elements are both covered together, or when either is covered.
- Tracking compound profile elements often require probing seemingly unrelated elements. For example, tracking when a variable is defined at location A and used in location B requires detecting any redefinition of the variable at other locations.

The rest of this section is divided as follows. First, we detail these considerations and how they affect our design choices in the various components of the system architecture. Next, we present our experimental setup and findings.

3.4.1 Architecture

We describe the design choices in the two stages of our instrumentation:

1. **Generating failure detection rules**: Figure 3.1 shows how the detection rules are generated. In this stage, the subject program is first profiled for every test case. Next, the profiles are post-processed to characterize each test case with the various *structural* and/or *substate* elements it covers. Then, both coverage and failure information are processed by an Integer-Programming-based optimization engine to determine a list of rules that can detect failure. A list of weights (or costs) for the different types of profiling elements is also passed to the optimization engine to define which elements are less expensive to probe.

2. **Instrumenting the subject program**: The generation of the modified subject program is described in Figure 3.2. Compound profiling elements which are detected in the previous stage often require multiple probes to profile and can sometime share some of these probes with another profiling element. The process of determining the individual probes is described in the following subsections. Figure 3.3 illustrates the modifications that are made on the subject program. In addition to the probes, a verifier is inserted to check if the run matches the generated rules.

Figure 3.1: Generation of failure detection rules. After profiling all subject program runs, clustering is performed, and each run gets associated with the clusters it covers. Finally, an optimization engine determines which combination of clusters can predict.

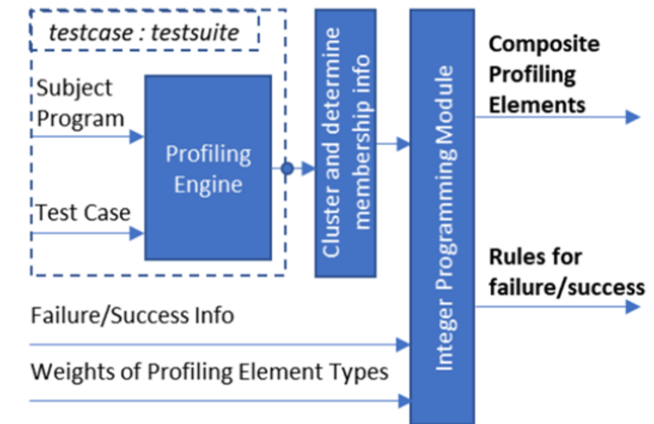


Figure 3.2: Generating the modified subject program. We extract a set of simple profiling elements that can help generate these statistics. The information about the profiling elements and the rules to be used for detection are passed to the instrumenter, which in turn generates a modified subject program.

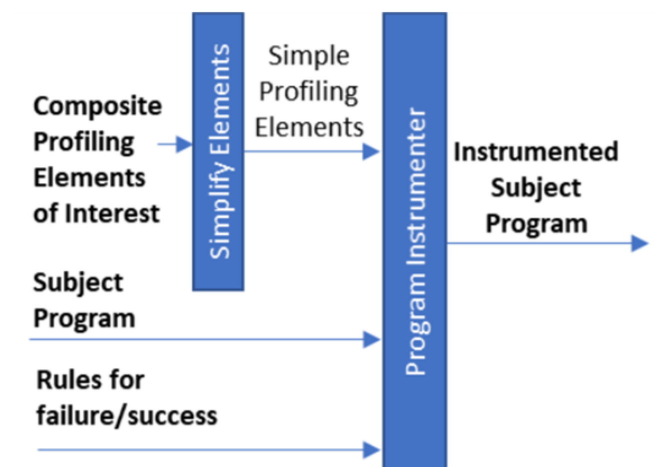
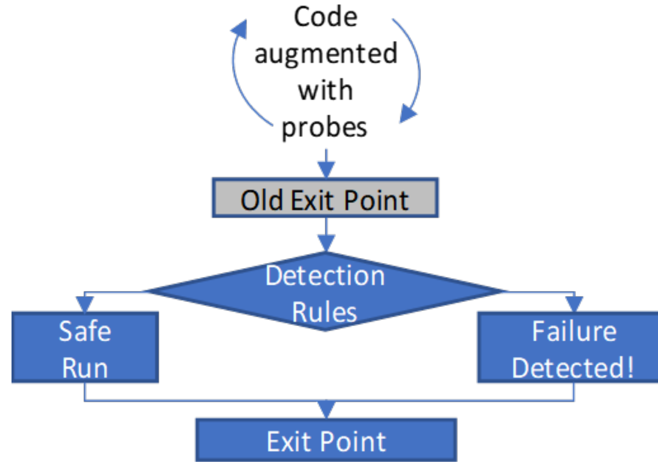


Figure 3.3: Modified subject program with failure detection. The original subject program logic is used, with two additional modifications: (1) probes are inserted at selected locations, and (2) before exiting, the program checks if failure detection rules are met.



Full Instrumentation and Profiling

The subject programs are first instrumented in order to generate the profiles needed to produce the failure detection rules. We provide in this subsection an overview about *structural* profiling and *substate* profiling.

Simple structural elements can be detected using a simple probe. For instance, *BB*s are detected by inserting a probe at the beginning of a basic block. On the other hand, compound elements such as *BBE* are detected by tracking the order in which the probes are triggered. Namely, the pair of the last two basic blocks that are triggered identifies a *BBE*. Similarly, *def-use pairs* are determined by keeping track of the last location each variable was defined in the program: this instance is then combined with the location it was used in to form the *def-use* pair. Further state information should be tracked as well such as the current method under execution for every thread. We refer the reader to [13] and [14] for further details about the methodology used.

Likewise, *substate* information is determined by adding a probe at the definition of each variable (i.e. local variable, static field, and instance field). The probe tracks the different values that the variable takes for a run and extracts a set of 17 statistical metrics (e.g. min, max, average, variance). Detailed information can be found in [13] and [14].

Clustering and Membership

In the training stage, determining *structural* coverage for a test case is straightforward: the *structural* element is covered when a probe or a combination of probes are visited. Each test case can then be associated with a binary vector, where 1 indicates that the element was covered, and 0 indicates the converse.

On the other hand, determining *substate* coverage is more complicated. The metrics that are generated for each *Cp* might be unique for each run (i.e. test case). Instead, for a given *Cp*, the test cases are clustered into multiple clusters, such that intra-cluster distances are minimized and inter-cluster distances are maximized. The clustering technique used is k-means [45], with two additional enhancements: (1) k-means++ [46] to select the initial cluster configuration, and (2) the elbow method to determine the optimal value of K (i.e. number of clusters). More details of the methodology can be found in [14]. Each test case can now be associated with a single cluster for each *Cp*. This membership information is captured in a binary vector similar to the one described above for structural profiles.

Note that we do not use failure information to determine the clusters. This prevents overfit associations between failures and the generated clusters in the next stages. In other words, this process ensures that a *substate* descriptor is general and robust enough to encompass multiple test cases, and unobserved

data.

Both *structural* and *substate* vectors can be combined before the optimization is performed. It should be noted that relevant cluster information needs to be saved at the training stage to be retrieved at the detection stage.

Optimization

For each subject program, we attempt to associate failure with one or more profiling elements. The binary feature vectors that were collected at the previous stage can be combined to form a matrix $\mathbf{M}$, where each row consists of the profiling elements covered by a single test case. Each row is then associated with the failure info of the test case: if it failed or not. Determining which elements are associated with failure can then be reduced to a Machine Learning or an Optimization problem. In this work, we consider Integer Programming.

Consider the example in Table 3.4 illustrating the membership matrix of a subject program, along with the failure data for four test cases. In this example, five profiling elements are considered ($\mathbf{M}_{.,1} - \mathbf{M}_{.,5}$). Note that no single element can fully predict a failure in this example. A possible way to predict failure is determining when the elements $\mathbf{M}_{.,3}$ *AND* $\mathbf{M}_{.,5}$ are covered. To generate similar solutions, consider the following Integer Programming rules.

For a given feature matrix $\mathbf{M}$, we denote by $\mathbf{M}_{\boldsymbol{tc,eid}}$ a matrix element, where tc is the test case number, and eid is the profile element id. We denote by F the failure data vector.

Also, consider a binary vector $\mathbf{V}$ describing which elements approximate the failure data. In other words, $\mathbf{V}_{\boldsymbol{eid}}$ is 1 when the element eid should be selected.

To satisfy the *AND* constraints, the following inequations should hold.

Table 3.4: Table illustrating when AND can be used to detect failure

Test ID	$\mathbf{M}_{.,1}$	$\mathbf{M}_{.,2}$	$\mathbf{M}_{.,3}$	$\mathbf{M}_{.,4}$	$\mathbf{M}_{.,5}$	Fail?
1	1	0	0	1	0	0
2	1	0	**1**	0	**1**	1
3	1	0	1	0	0	0
4	1	0	0	0	1	0

$$\begin{cases} \mathbf{M} \times \mathbf{V} \geqslant |\mathbf{V}| \times \mathbf{F} \\ \mathbf{M} \times \mathbf{V} \leqslant (2\mathbf{F} - 1)\epsilon + |\mathbf{V}| \end{cases} \tag{3.1}$$

, where ϵ is an infinitesimally small positive number. Consider the two inequations for a single test case. The product $\mathbf{M}_{tc} \times \mathbf{V}$ evaluates to the number of selected profile elements that were covered in a single test case. The product $\mathbf{M}_{tc} \times \mathbf{V}$ to be greater or equal to the weight of $\mathbf{V}$ when the test case fails, and strictly smaller than the weight of $\mathbf{V}$ when the run is passing. The first inequation achieves the former goal, while the second one achieves the latter goal. The entity $(2\mathbf{F}_{tc} - 1)\epsilon$ is equal to ϵ when $\mathbf{F}_{tc} = 1$, and $-\epsilon$ otherwise. These two inequations, along with the fact that $\mathbf{V}_{eid} \in \{0, 1\}$ ensures that $\mathbf{M} \times \mathbf{V}$ is equal to $|\mathbf{V}|$ only for failing test cases, and is necessarily smaller than $|\mathbf{V}|$ for passing test cases.

A similar analysis can be made for the *OR* operator. In the example in Table 3.5, note how $\mathbf{M}_{.,3}$ and $\mathbf{M}_{.,5}$ can predict failure together. A test case failure occurs when any one of these elements is covered.

The constraints can be represented in the following inequations.

$$\begin{cases} \mathbf{M} \times \mathbf{V} \geqslant \mathbf{F} \\ \epsilon \times \mathbf{M} \times \mathbf{V} \leqslant \mathbf{F} \end{cases} \tag{3.2}$$

Table 3.5: Table illustrating when OR can be used to detect failure

TEST ID	$M_{.,1}$	$M_{.,2}$	$M_{.,3}$	$M_{.,4}$	$M_{.,5}$	FAIL?
1	1	0	0	1	0	0
2	1	0	**1**	0	**1**	1
3	1	0	**1**	0	0	1
4	1	0	0	0	**1**	1

The first inequation ensures that $\mathbf{M}_{tc} \times \mathbf{V}$ is strictly greater than zero, when $\mathbf{F}_{tc} = 1$, whereas the second inequation ensures that $\mathbf{M}_{tc} \times \mathbf{V} < 1$ (effectively equal to 0) when $\mathbf{F}_{tc} = 0$.

In addition to the feasibility constraints, we aim to minimize the number of elements selected in $\mathbf{V}$ (formally, $min \sum \mathbf{V}_i$). More elaborate objectives can consist of minimizing the estimated number of probes, or to favor simple profiling elements. For instance, compound *structural* elements need more probes than simple *structural* elements.

Without loss of generality, the improved objective function is to minimize the scalar product $\mathbf{V}.\mathbf{W}$, where $\mathbf{W}$ is a weighted vector that specifies the cost of each profiling element. If all elements are equally expensive to profile, then $\mathbf{W} = [1, ..., 1]^{\mathsf{T}}$. In the experimental setup, we denote by *EqualWeight* the configuration where the cost of all elements is equal to one, and we denote by *PreferBB* the configuration where the cost of *DUP* and *BBE* elements is equal to 4, and the cost of *BB* elements is equal to 1.

Selective Instrumentation and Profiling

To enable fast online failure detection, the profiling cost should be minimized. Therefore, we make some changes to the instrumenters/profilers to profile only the elements discussed above. Each *substate* and simple *structural* element is associated with one probe. Therefore, determining which probes to insert is

straightforward when these types of elements are included in the generated detection rules.

For compound *structural* elements, which consist of a (source, destination) pair, two probes are usually not enough to characterize the element. Consider the following two examples: (1) on *BBE* and (2) on *DU*.

Consider the basic blocks A, B and C, and consider the following possible edges: A-B, A-C and B-C. To detect when the edge A-C occurs, inserting probes at A and C is not enough. If A was visited first, followed by B then C, the probes would show that A-C was covered, when, in fact, A-B and B-C were covered.

A similar scenario can occur with *def-use* pairs. To detect when a variable, $\mathfrak{v}$ is defined at location $\mathfrak{l}_1$ and used at location $\mathfrak{l}_3$, the variable $\mathfrak{v}$ should not have been redefined at other locations. To guarantee that, we need to insert probes at all the possible definition locations of $\mathfrak{v}$ that can be used at the destination location.

In summary, a compound structural element consisting of a (*source*, *destination*) pair (i.e. *BBE* or *DU*) require more than two probes to capture events that invalidate this pair. On the other hand, a probe can serve to detect multiple compound profile elements. The combination of these two conditions is used to minimize the number of probes inserted for failure detection.

3.4.2 Experimental Setup and Results

We evaluate our proposed method on *Defects4J* [15], a Java framework used extensively in software research. Six open-source libraries with their various versions were used in our evaluation. Each library version is associated with a single bug and has a *Test Suite* that consists of thousands of passing test cases and a very small number of failing test cases. The libraries and the number of

versions used are described below.

- *JFreeChart* with 26 versions
- *Joda-Time* with 27 versions
- *Commons Lang* with 65 versions
- *Commons Math* with 106 versions
- *Closure Compiler* with 133 versions

For a given version, *substate* and *structural* profiles are generated for each test case. Then, the test suite profiles are analyzed using the Integer Programming approach described in Section 3.4.1 for either the *OR* or the *AND* operator and a selected **W** vector. When not defined explicitly, the **W** vector is the identity vector.

The results are illustrated as follows. First, we present an analysis about the number of subject programs with a feasible solution. Next, we analyze the number of profiling elements necessary in the trained model. Then, we discuss the resources required by our method and some of the additional optimizations performed. Afterwards, we analyze the performance of the trained models when the *Test Suite* is re-run. Finally, we analyze the performance of our proposed method when we vary the arithmetic operator or we select a different weight **W**.

Training the Failure Detection Model

We profiled each *subject program* and generated matrices based on its *structural* and *substate* profiles, as described in Section 3.4.1. Table 3.6 shows the number of versions per library that ended in a successful failure detection training, using

Table 3.6: Failure detection accuracy for different profile types

Program	|Versions|	Structural	Substate	Hybrid	Structural and Substate	Structural and ¬ Substate	¬ Structural and Substate	No Solution
Chart	26	24	20	25	19	5	1	1
Time	20	18	17	19	16	2	1	1
Lang	62	39	52	56	36	3	16	6
Math	96	71	86	90	67	4	19	6
Total	204	152	175	190	138	14	37	14

the *OR* operator. The first two columns list the libraries and the number of faulty versions that were tested. Versions that failed during testing due to lack of resources during profiling or training were ignored completely for the purpose of this experiment. The remaining columns show the number of faulty versions with a feasible solution, given the type of profiles used. Namely, the types that were considered were: *structural* profiles, *substate* profiles, and the combination of both, *hybrid*. On the right, we present the number versions where training was successful for a given profiling type, but not the other. Last, we show the number of versions where the optimization failed to generate a valid solution.

These results show that *structural* and *substate* profiles can characterize failure in 75% and 86% of versions respectively, and combining both profiles can characterize 93% of versions. This result confirms the complementary nature of these profiling types. A deeper analysis shows that, in *Chart* and *Time*, the feasibility rate was higher for *structural* profiles, whereas *Lang* and *Math* showed a higher accuracy for *substate* profiles. Additionally, for these two libraries, *substate* profiles exclusively contributed to 21% and 29% of all solutions, in contrast to 4% and 5% for *Chart* and *Time*. This result indicates that the faults present in the first two libraries are *structural* in nature, while the faults in the remaining libraries are data dependent. A similar result was found when performing Fault

Localization using *structural* profiles in [14].

Considering all programs, failures could be characterized with feasible models in 14 versions when *structural* profiles was used exclusively, and in 37 versions when *substate* was used exclusively, which highlights a complementary nature of *substate* and *structural* profiles. A large number of versions had feasible solutions for either profile types, which indicates that failures in these versions leave distinct profiles in both data and structural subspaces.

Number of Profiling Elements

The number of profile elements can indicate the health of the trained models. In Table 3.7, we report the number of profile elements, which, when merged using the *OR* operator, can detect failure at runtime. For instance, in 20 versions of *Chart*, detecting failure can be performed using a single *structural* profiling element. Only 4 versions require more than one *structural* profiling element to detect failure. The remaining libraries show one or two elements required to predict failure in any of their versions. This shows that the suggested profile element likely does not overfit the test suite.

Note that the reported numbers do not necessarily correspond to the number of probes. While a simple *structural* profile element such as a *basic block* or a *substate* profile element only require one probe, compound *structural* profile elements can require multiple probes. To effectively minimize the number of probes, additional modifications should be added to the optimization model. These modifications are presented in the experiments' discussion section.

Table 3.7: Distribution of the number of profile elements to be combined with the OR operation

Program	#Versions	#Elements	Structural	Substate	Hybrid
Chart	26	1	20	17	18
		2	1		
		4	2	2	2
		5	1		1
Time	27	1	16	16	18
		2	1	2	1
Lang	65	1	34	48	50
		2	5	4	12
Math	106	1	62	70	77
		2	9	32	26

For each program, and profiling type, the number of profile elements, which are combined with the OR operator at runtime, is reported.

Resource Usage

We ran our experiments on a Windows 10 Virtual Machine with 100 GB of RAM and 12 *vCPUs* of 2.0 GHz each, and an SSD disk. Many of these experiments were run in parallel, while maintaining non-saturated CPU and RAM usages to ensure reliable timing and resource usage metrics.

The profiling time, which was reported in [14], ranges between a couple of minutes for small programs like *Chart* and more than 3 hours for programs like *Math* or *Closure*. Although these numbers are high, profiling can be parallelized and further optimized if needed in a real-life scenario.

However, the time taken during failure detection training, which cannot be easily parallelizable, is reported in Table 3.8. As can be seen, the time taken ranges between a minute and 280 minutes for all programs. On the other hand, Table 3.9 shows the required memory for each version. *Closure* had the highest

Table 3.8: Range of time taken during the optimization process per program version (in minutes)

Program	Structural	Substate	Hybrid
Chart	[1, 20]	[0, 38]	[1, 20]
Time	[2, 80]	[0, 5]	[1, 86]
Lang	[0, 10]	[0, 2]	[0, 25]
Math	[0, 6]	[0, 280]	[0, 140]

Table 3.9: Average memory used in megabytes during the optimization process

Program	Structural (MB)	Substate (MB)	Hybrid (MB)
Chart	2,722 ± 1,077	480 ± 198	5,336 ± 1,387
Time	4,320 ± 511	705 ± 253	8,443 ± 586
Lang	904 ± 404	322 ± 105	1,646 ± 974
Math	5,954 ± 3,711	1,184 ± 761	11,243 ± 7,725

memory and time requirements and consumed more than 100 GB of RAM, which called for additional optimizations.

To minimize the time and resources required during the optimization of *Closure* versions, the following feature pruning was performed. Before performing an *OR* optimization, profile elements that clearly do not contribute to the final solution were deleted. Formally, all profile elements that were covered during a passing test case are removed. A solution that uses these elements cannot exist, since it would erroneously classify a passing test as failing. Similarly, for *AND* combinations, any profile elements that are not covered in a failing run are removed. A solution that includes such elements cannot exist, since it would wrongly classify a failure to be a passing test.

After performing these additional optimizations, the number of profile elements was reduced from the range of tens of thousands to hundreds. This marginally reduced the time and memory required to perform training to under two hours and 1 GB. We consider these values acceptable for the scope of our

method.

Testing the model

For each library version that resulted in a feasible model, we store in a *definitions* file the operator used (*OR* or *AND*) and explicit details about the profile elements determined in the optimization. Namely, for structural elements, we extract the descriptors of the *BB*, *BBE*, or *DU*, which are determined to be predict failure, along with the descriptors of complementary *BBE*s and *DU*s that can invalidate these elements, as described in Section 3.4.1. Similarly, for *substate* profiles, each profile element refers to one of many clusters belonging to a single *Cp*. Therefore, the following information needs to be saved for each profiling element:

- *Cp* information, namely, the definition location of a variable
- The minima and maxima vectors used in the normalization of the features before clustering
- All centroids belonging to this *Cp*

In the selective instrumentation phase, we determine the probes needed for each profile element. Each *substate* profile element is associated with a single probe and is straightforward to profile. On the other hand, the structural instrumentation algorithm was modified as follows. The addition of each probe was guarded by a set of rules to judge if a probe should be added or not. Some complex rules included complicated tests such as the ones for adding a probe at the entry of a *method*. Determining which method the program is currently in is essential to all three types of *structural* profiles studied. To determine if a method should be probed, the method should have at least one of the following: a *BB* determined to predict failure inside that method, a *BBE* within that method or

that has this method as a source or destination, a *DUP* that has either its *Def* or *Use* within that method, or if it is part of a *DUP* or *BBE invalidator*.

In addition to the above, the program is augmented to load the rules at the beginning and test them before exiting, as discussed in Figure 3.2. When tested on 3 libraries, *Chart*, *Time*, and *Lang*. Out of 83 versions that had feasible solutions from *structural* profiles, 79 versions accurately predicted failure without any false positives or false negatives. These numbers were the same for both *EqualWeight* and *PreferBB* weights, although different versions missed. For *substate* profiles, out of 91 versions having feasible solutions, 90 had no False Positive and no False Negative. These numbers give confidence in the modifications performed on the instrumenters and profilers, and in the detection approach in general.

At runtime, we collect the increased times during failure detection for all profiling types. The program augmented with failure detection capabilities using *Substate* profiles had a negligible increase in elapsed time (less than 1%), whereas *structural* profiles with weights *EqualWeight* and *PreferBB* were 12 and 8 times slower, respectively. Although these numbers are high, note the time decrease for the *PreferBB* method, which selects *BB* in favor of compound *structural* elements with 4:1 odds. Note that these high timing values can be contributed to multiple factors, including the type of probes we used, which rely on expensive Java constructs, such as *Throwable*'s to determine the stack trace, and the high number of probes per profile element, especially for *def-use* profile elements. Several techniques can be used to marginally reduce these numbers.

3.5 Analysis

In this chapter, we presented *D-FUSE*, Detecting Failure Using Substate Profile Elements, a technique to detect failures using *substate* and *structural* profiles. Our framework enables a *subject program* to signal when a failure is imminent. In the experimental section, we show that *substate* profiles complement *structural* profiles in vulnerability detection. Our proposed method relies on an Integer Programming optimization technique to select a combination of profiling elements that detects the execution of a vulnerability, while minimizing the total number of profiling elements. We tested our framework on *Defects4J* and showed that our proposed method characterized a vulnerability in 93% of the profiled programs. Out of these versions with characterized vulnerabilities, only 4.8% showed an incorrect classification. These cases can be mainly attributed to the randomized data in the *Test Suites*. Furthermore, we show that our method is robust both quantitatively and qualitatively. Quantitatively, the number of profile elements involved in a single version is limited to 1 or 2 for the majority of versions. On the other hand, favoring simple *structural* elements and clustering *substate* elements without access to failure prevents overfitting.

Moreover, our framework has flexible parameters used in the objective function. We showed how the framework can be used to promote simple profiling elements over compounded elements. Other objective functions can be explored. Advanced weighting can include assigning effective weights relative to the number of probes or to the number of times these probes would be triggered on average. Other directions that this work can take is to favor elements that occur earlier in the program, and promote early detection. This can be achieved by calculating the average execution distance from each probe to the entry point of a library.

Chapter 4

Related Work

4.1 Related to Repair

We present in this chapter our preliminary literature review, as it will appear in [5]. Zhang et al. [47] presented a fault localization technique that is very relevant to our patch generation approach. It entails switching the valuation of the program's predicates, each one at a time for the purpose of producing the correct behavior. A predicate switch that yields a successful program completion can be further analyzed in order to identify the cause of the defect. Our approach differs in that: 1) due to our accurate CBFL technique, only few predicates need to be explored for switching; 2) predicate switching is considered at execution instances discovered by our approach; and most importantly, 3) a code patch is provided.

SPR [32] performs staged program repair. It performs fault localization using frequency analysis of positive and negative test case coverage. It leverages a set of parameterized transformation schemas (PTS) each of which targets a class of defects. For each PTS, SPR searches for an evaluation of schema parameters that

allow the schema to produce a successful repair. It dismisses the transformation schemas (and their repairs) that fail the target value search and proceeds. For example

PTS "if(cond || abstract_cond())" refines predicate "if(cond)" and "abstract_cond()" can return either true or false.

If both target values do not fix the defect, the PTS dismisses. The last stage is condition synthesis where SPR uses the constraints obtained from the target value search to synthesize a condition. In particular, SPR selects states of program variables that are invariant for positive test cases (PPred), and states of program variables that were invariant for negative test cases (NPred). The latter are invariants that hold while the target value succeeds at fixing program behavior. The PTS synthesizes the condition such that the target values are obtained when NPred hold and PPRed don't - "(!PPred) implies NPred". CFAAR is similar to SPR in that it uses both positive and negative test cases to synthesize a fixing condition. However, SPR performs a search for fixes matching existing schemas and consequently it has to use the values to determine the search rather than guide the search. CFAAR uses a classifier to determine whether a predicate needs to be changed and then uses the successful sequence of modified values to deploy a dynamic fix and synthesize a patch.

Precise condition synthesis [33] presents ACS to solve the overfitting problem in automatic defect repair. ACS splits the problem into (1) selecting the variables to be included in the precise condition, and (2) selecting the predicate from a set of existing predicates. It uses dependency order and analysis of API comments provided in javadocs to rank and select the variables, and uses predicate frequency in similar contexts to rank and select the predicates. CFAAR shares with ACS that it looks for more precise predicates to solve the overfitting prob-

lem. However, CFAAR inspects an infinite possible search space and uses tests to guide the search while ACS is restricted to existing code fragments. Genesis [48] infers new patches from existing patches to fix (1) null pointer, (2) out of bounds, and (3) class cast defects. In a sense it refines the search space of [6] and [49] to concentrate on successful human patches instead of general code fragments, then it expands the potential search space by inferring transforms that generate defect patches from the existing patches. A transform is specified with two abstract syntax trees; one matches the faulty fragment in the original code, and the second specifies the replacement code. Generators allow introducing new logic and design elements in the fix specifically for template variables that are not matched in the code. Integer linear programming (ILP) is used to limit the search space to a reasonable number of patches by maximizing the number of training patches that cover the inferred search space. Unlike ACS and SSCR, Genesis is not limited to existing code fragments. However, it is limited to patches that are syntactically related to the existing patches through an AST. CFAAR differs in that it inspects a search space that is related semantically to the defect and uses the test cases to guide the search. Semantic search for code repairs (SSCR) [6] characterizes a large set of code fragments with FOL constraints and considers those potential fixes, relates a fault in a program to fragments of code in the program and characterizes these fragments with fault FOL constraints, then it uses constrain solvers to match the fault constraints with the potential fix constraints. The technique finally integrates the selected fix by syntactic modifications such as renaming variables. CFAAR is similar in the sense that it also performs a semantic search, it differs in the sense that it is not limited to a large set of existing code fragments. CFAAR searches an infinite space of potential fixes that is the composition of modifications of failing control statements and

uses test cases to guide the search. While the work in [6] relies heavily on computationally expensive SMT solvers, CFAAR leverages decision tree algorithms as heuristics to construct the fix. Le Goues et al. [7] proposed GenProg, a repair technique based on genetic programming. They assume that repairing a fault in one function can make use of snippets of code appearing in other functions in the program. For example, several existing functions in a program might implement checks for whether a pointer is null, the corresponding code can then be inserted in the function under repair in the aim of repairing it. The technique explores different variations of the defective program such as those resulting from inserting statements, deleting statements, and swapping statements. Also, mutation and crossover operators are applied and guided using a fitness function that evaluates the generated program against the test suite. Once a repair is found, it is further refined using delta debugging by discarding the unnecessary statements within. Our repair technique is very different in terms of its underlying approach and the nature of the produced solution. Assiri and Bieman [8] evaluated the impact of ten existing CBFL techniques on program repair. Specifically, they measured their impact on the effectiveness, performance, and repair correctness of a brute force program repair tool, i.e., a tool that exhaustively applies all possible changes to the program until a repair is found. A brute-force repair tool is guaranteed to fix a fault if a repair is feasible. Therefore, a failure to find a potential repair would likely be related to the selected CBFL technique. Including our proposed CBFL technique in their comparative evaluation would be valuable, as it could help justify its cost. Martinez and Monperrus [50] presented Astor, a library comprising the implementation of three major program repair approaches for the Java platform. The library is also meant to be extended by the research community by adding new repair operators and approaches. The currently supported

approaches that originally targeted C programs are: 1) jGenProg2: an implementation of GenProg for Java [25] [7] in which repair operators only consider nearby code, and not the whole codebase as it is the case in GenProg. 2) jKali: an implementation of the Kali approach [51] for Java, which performs repair by exhaustively removing statements, inserting return statements, and switching predicates. Our approach is far from being exhaustive since the predicate switching is highly targeted in terms of location and time. 3) jMutRepair: an implementation of the approach presented by Debroy and Wong [31] for the java platform. jMutRepair mutates the relational and logic operators in suspicious if condition statements. Since our approach negates predicates at the byte code level (single clause predicates), it practically also mutates relational and logic operators. However, unlike jMutRepair, our approach negates the predicates at specific execution instances.

Nopol [26, 34] uses angelic fix localization to locate faulty if-then-else conditions, execute passing test cases to compute a model of the correct behavior of the program, abstract the values of the variables in the model to FOL constraints, and uses SMT solvers to compute a fix to the condition. The technique targets buggy if conditions and missing preconditions. CFAAR is similar to Nopol in that it uses both the passing and the failing test cases to compute a model for the fix. CFAAR differs in the abstraction techniques as it is variable specific while Nopol creates SMT statements to model the execution. Finally, CFAAR uses decision trees to compute a potential code fix, while Nopol uses SMT solvers. An influencing precursor of Nopol is SemFix, an approach presented by Nguyen et al. [52]. SemFix is based on symbolic execution, constraint solving, and program synthesis. Given a candidate repair location l, SemFix derives constraints on the expression to be injected at l in order to make the failing test case pass.

Symbolic execution is used to generate the repair constraints, and program synthesis is used to generate the repair patch. Similar to SemFix, DirectFix [30] and Angelix [29] also aim at synthesizing repairs using symbolic execution and constraint solving; but are more scalable. Tan and Roychoudhury [53] presented relifix an approach for repairing regression bugs. The mutation operators considered are derived by manually inspecting real regressions bugs. The potential repair locations were identified by differencing the current version of the defective program with its previous version, and by considering the Ochiai suspiciousness of the locations. Pei et al. [54] proposed an approach that exploits contracts such as pre/post-conditions to localize faults and generate repairs in Eiffel programs.

Elkarablieh and Khurshid [27] developed a tool called Juzi, within which the user provides a Boolean function that checks whether a given data structure is in a good state. The function is invoked at runtime, and in case a corrupt state is detected, the tool performs repair actions via symbolic execution. One of the authors later targeted the repair of the selection conditions in SQL select statements [55].

4.2 Failure Detection as an Intrusion Detection System

Our work, *D-FUSE*, which was presented in Chapter 3, extends the work in [11] and [56] with *substate* profiles. The original techniques applied in these works perform signature matching using different types of *structural* profiles, whereas our novel technique combines both *structural* profiles and *substate* profiles for intrusion detection. It also enables the developer to prioritize certain types of profile elements over others for more robustness, better interpretability and lower

resource overhead. The rest of this subsection discusses related work on Intrusion Detection Systems (IDS), which can be divided into the following categories, by decreasing order of relevance to our work: Profile-based, Pattern-matching, Taint-based, and Anomaly-based.

Profile-based techniques include PQL [57], which derives a vulnerability signature with the assistance of the developer. The proposed system enables the developer to generate the signature by inspecting the subject program code and by providing a combination of events that characterize the vulnerability. ASP [58] is used to instrument the subject program and to generate the execution traces. Events can include combinations of method invocations, field accesses and error descriptions. The system then monitors if these events are executed at runtime. Another profile-based technique is the one proposed by Lorenzoli et al. [59], which allows the developer to instrument relevant program points in order to abstract invariants using *Daikon* [60] and characterize the vulnerability. These program points are then monitored at runtime, and an intrusion is detected when these invariants are violated.

Less relevant techniques for vulnerability detection include pattern-matching, taint-based, and anomaly-based techniques. Pattern-matching techniques originally focused on characterizing *exploits* (payloads that exploit a vulnerability) [61], but shifted towards characterizing the vulnerability itself, such as in [62], to avoid *polymorphic* attacks. Brumley et al. [63] proposed to use formal static analysis to generate a vulnerability signature. Data mining techniques such as Classification, Association Rules, and Frequent Episodes were used in [64] to analyze the vulnerabilities.

Taint-based techniques such as *TaintCheck* [65] and *Panorama* [66], allow the user to mark (taint) inputs and track them using dynamic data-flow anal-

ysis. Anomaly-based techniques, which intend to generalize to novel attacks, characterize normal behavior and detect deviations from that behavior. Some of these techniques rely on analyzing calls (stack calls[67] or system calls [68][69]). Other approaches proposed static analysis such as in [63] and [62]. Contrary to anomaly-based approaches, our suggested approach closely relates to signature-based approaches, which focus on detecting known types of vulnerabilities.

Chapter 5

Conclusion

In this thesis, we introduced two novel methods that can be used to mitigate the costly effect of discovered bugs in production, before a fix is released. Both methods leverage the existing *Test Suite* associated with the subject program to create a patch. In the first method, which is an Automated Program Repair method, we attempt to produce a patch that makes the *Test Suite* pass. We also proposed *D-FUSE*, which is a failure detection technique that produces a patch that detects when a failure is imminent and sends a signal to a controller that takes the appropriate measure, such as reverting a transaction, disconnecting the user, or raising a critical ticket to the operations and development teams.

The proposed APR method, *CFAAR*, works as follows. It is a test-based repair assistance technique that targets defects that are repairable by negating control statements under some specific conditions. *CFAAR* relies on the program's state to determine when a candidate control statement should be negated in order to yield correct behavior. Namely, for a given subject program, it determines if switching an *if-statement* makes one of the failing test cases pass. If it does, *CFAAR* profiles all accessible state and applies a *Decision Tree* algorithm to

determine the proper rules and conditions to switch the value of the *if-statement*. The synthesized patch can be presented to the developer for further evaluation, or applied directly to the program at the *bytecode* level, if deemed correct.

Our experiments involving 149 defects revealed the following: 1) 91 defects were found to be potentially repairable by *CFAAR*; 2) 41 plausible patches were generated by *CFAAR*; and 3) at least 12 plausible patches are believed to be correct. In the future, experiments can be conducted to assess the level of repair assistance our plausible patches provide to the developer.

Our second proposed method, *D-FUSE*, relies on both *structural* and *substate* profile information to generate a minimally sufficient set of profiling probes that can predict when a failure is imminent. The process relies on a Machine Learning model that optimally selects a combination of profile elements. We also proposed proof-of-concept extensions to our work:

- Using a different operator (*AND* and *OR*) to combine profile elements
- A cost-per-element-type weighting, which can favor certain types of probes that are inexpensive to profile
- A cost-per-element-instance weighting, which can favor probes at known locations that are inexpensive to profile

When tested on a major library, we were able to generate accurate failure oracles for 93% of the versions, compared to only 74.5% when using *structural* profiles. Furthermore, our method was associated with an acceptable overhead.

Appendix A

Complementary Material for the Repair Method

This appendix complements the evaluation section of the repair method. This document is a work in progress, and is divided as follows. The first section presents an analysis of a sample of patches generated by testing all negation patterns on predicates identified as suspicious using CBFL, whereas the second and last section presents an analysis of the patches generated by considering predicates that we manually identified as suspicious. Out of the 28 programs that we manually selected from the Siemens dataset, 1 still require further analysis. Out of the remaining 27 programs, we found a plausible patch for 24 programs, a correct fix for 8 programs, and a near-correct fix for 2 programs. Out of the programs considered in both datasets, we verify manually that we generated 12 correct patches:

- For Siemens, we generated 8 correct patches for print_tokens2 v7, print_tokens2 v8, print_tokens2 v9, schedule v4, replace v7, replace v16, replace v28, and replace v30.

- For IntroClass, we generated 4 correct patches for syllables v1, syllables v2, grade v13, and digits v19.

Please note that $\wedge$ indicates an XOR operation.

A.1 Analysis of the Patches Generated on Suspicious Predicates

A.1.1 Sample Correct Patch 1: print_tokens2 - v8

Original Buggy Code

```
boolean is_token_end(int id,char ch) {
        if (id == 1) {
                ...
        } else if (id == 2) {
                ...
        } else {
                if(ch ==' ' || ch=='\n' || ch==59 || ch == '\t')
                        return T;
                        //fault -- added "|| ch == '\t'"
                ...
        }
}
```

```
...
while (is_token_end(id,ch) == F) {
        ...
}
...
```

Patched Code

```
...
while ((is_token_end(id,ch)) == F ^ (id <= 0 && ch = 9)) ...
```

Discussion (show decision tree).

The corrected code will negate (xor) the condition provided by is_token_end when: id ¡= 0 and ch = 9 (which is '

t'). Although Fault Localization did not label the predicate ch == "

t" as suspicious, the classifier detected that negating the output of is_token_end when id ¡= 0 and ch = '

t' fixes the output.

A.1.2 Sample Correct Patch 2: syllables - v1 (v2 is very similar)

Original Buggy Code

```
if(ch=='a' || ch=='e' || ch=='o' || ch=='u' ||ch=='y'){
                    vowels++;
            }
        // fault --- missing || ch=='i'
```

```
if(ch=='a' || ch=='e' || ch=='o' || ch=='u' || ch=='y' ^ (ch!='y'
    ↪ && ch=='i')){
                        vowels++;
                }
```

The patch will negate the clause ch=='y' only when it's valuation is false and ch= 'i'. This patch is equivalent to the correct code.

A.1.3 Sample Correct Patch 3: grade - v13

Original Buggy Code

```
if (score >= a){
result += 'A';
}
else if ((score >= b) && (score > a)){
 // Fault can be fixed by removing score > a
 // or by putting score < a
result += 'B';
}
else if ((score >= c) && (score < b)){
...
```

Patched Code

```
else if ((score >= b) && ( (score > a) ^ true))
```

The patch will negate the clause (score ¿ a) which is always false, and therefore, the clause is bypassed.

A.1.4 Sample Correct Patch 4: digits- v19

Original Buggy Code

```
while (Num < 0)
{
        X = Num % 10;
        NewNum = (Num - X)/10;
        if ((X<0)) //Can be fixed by replacing with NewNum != 0
        {
                result += (X*-1);
                result += "\n";
        }
        if (Num < 0 && Num > -10)
        {
                result += X;
                result += "\n";
        }
        Num = NewNum;
}
```

Patched Code

```
if( (X < 0) ^ (X > -1 || NewNum > -1) )
```

In the context of this program, the following properties always hold:

```
X <= 0
NewNum <= 0
```

Therefore, the patched code is equivalent to:

```
if( (X != 0) ^ (X == 0 || NewNum == 0) )


\rightarrow ((X == 0) && (X == 0 || NewNum == 0)) || ((X != 0) &&
    ↪ (x != 0 && NewNum != 0))
\rightarrow (X == 0) || (X != 0) && NewNum != 0)
\rightarrow (X == 0) || (NewNum != 0)
\rightarrow (NewNum != 0) || (NewNum == 0 && X == 0)
\rightarrow (NewNum != 0)
```

since (NewNum == 0 && X == 0) → Num = 0 which is impossible since Num ¡ 0 is a precondition.

Therefore the patch is equivalent to replacing X¡0 by NewNum != 0. (correct fix)

A.1.5 Sample Incorrect Patch - tcas - v1

Original Buggy Code

```
//In: main() ->alt_sep_test() -> Non_Crossing_Biased_Climb():
...
     if (upward_preferred == 1)
     {
      result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
          ↪ (!(Down_Separation > ALIM()))); //fault: should have
          ↪ >=ALIM
     }
```

Patched Code

```
//In: main() ->alt_sep_test() -> Non_Crossing_Biased_Climb():
...
    if (upward_preferred == 1)
    {
     result = !(Own_Below_Threat()) || ((tmp = Own_Below_Threat()
        ↪ ^ tmp > 0) && (!(Down_Separation > ALIM())));
    }
```

This patch is likely to be incorrect.

A.1.6 Sample Undecided Patch

Original Buggy Code

```
   switch(next_st)
    {
        default : break;
        case 6 : /* These are all KEYWORD cases. */
        case 9 :
        case 11 :
        case 13 :
        case 12 : //fault -- added case
        case 16 : ch=get_char(tstream_ptr.ch_stream);
            if(check_delimiter(ch)==T)
                {
                 token_ptr.token_id=keyword(next_st);
```

```
                    unget_char(ch,tstream_ptr.ch_stream);
                          token_ptr.token_string[0]='\0';
                          return token_ptr;
                   }
                   unget_char(ch,tstream_ptr.ch_stream);
                   break;
```

Patched Code

```
if((tmp = check_delimiter(ch))==T ^ (cu_state <= 0 && tmp > 0)) {
                    token_ptr.token_id=keyword(next_st);
                    unget_char(ch,tstream_ptr.ch_stream);
                          token_ptr.token_string[0]='\0';
                          return token_ptr;
}
```

Discussion This is equivalent to say that the if statement block is not executed if cu_state is ¡= 0. In that case, get_char then unget_char will be executed, which is equivalent to the fix. To verify that this fix is correct, we need to prove that cu_state ¡= 0 is equivalent to next_st = 12 or next_st != 6 , 9 , 11, 13, 12, 16.

We need formal verification to test if this is correct.

A.2 Patches generated on manually specified predicates

By manually specifying the suspicious predicate to negate, we obtained the results presented in the following 3 tables.

Table A.1: Summary of the patches generated for print_tokens, print_tokens2, schedule, schedule2

<table>
<tr><th>Program Version</th><th>Faulty code</th><th>Patched Code Analysis</th></tr>
<tr><td>print_tokens seedV7</td><td>while(check_delimiter(ch)==F)
{
if(token_ind >= 10)
//fault: should have 80 instead of 10
break;
token_str[token_ind++]=ch=get_char(tstream_ptr.ch_stream);
}</td><td>token_ind = 10

Not Plausible</td></tr>
<tr><td>print_tokens2 seedV7</td><td>if(ch=='\n' || ch==' ') //fault -- should have if(ch=='\n')
return T;
else
return F;</td><td>ch = 32

Correct</td></tr>
<tr><td>print_tokens2 seedV8</td><td>if(ch ==' ' || ch=='\n' || ch==59 || ch == '\t') return T; //fault -- added "|| ch == '\t'"</td><td>ch = 9
or always
or ch <= 9

The first one is correct</td></tr>
<tr><td>print_tokens2 seedV9</td><td>if(ch=='\n' || ch == '\t') //fault -- added "|| ch == '\t'"
return T;
else
return F;</td><td>ch = 9

Correct</td></tr>
<tr><td>schedule v4</td><td>if (count > 1) //fault: should be if(count > 0)
...</td><td>Switch when count = 1
Correct</td></tr>
<tr><td>schedule2 v7</td><td>if(ratio < 0.0 || ratio >= 1.0) //fault: should be ratio > 1.0
{
return BADRATIO;
}</td><td>Negating the predicate using the patterns did not fix any test case</td></tr>
</table>

Based on our analysis, some programs are most likely suffering from an incorrect or undecided patch due to the lack of diversity in the behaviors provided in the test suite. A good example illustrating this issue is tcas for which the code patches that were generated were plausible. While the code patches were compact, the patch consisted of removing or always negating the predicate in most cases, which fixed and preserved all available test cases.

Table A.2: Summary of the patches generated for tot_info

Program Version	Faulty code	Patched Code Analysis

tot_info v13	if (pj >= 0.0) //fault: should have > instead of >=
	info -= (pj * Math.log(pj));
	/* part 4 */
		switch when pj <= 0.0
		
		Near hit since pj can be negative

tot_info v14	if (r.value * c.value >= MAXTBL)
	//fault -- should have > instead of >=
	{
	myMethods.my_fputs("* table too large *\n");
	System.exit(0);
	}
		Always switch
		
		Likely incorrect

tot_info v15	if (Abs(del) < Abs(sum) * (EPS - 0.000001))
	//fault: should be: if (Abs(del) < Abs(sum)*EPS)
	return sum * Math.exp((x*(-1))
	+ (a*Math.log(x)) - LGamma(a));
		Requires further processing
		Out of memory

tot_info v16	if (info >= 0.1) //fault -- should have 0.0
	{
	Formatter f = new Formatter(System.out);
	f.format("2info = %5.2f\tdf = %2d\tq = %7.4f\n",info,
	infodf.value,QChiSq(info, infodf.value));
	totinfo += info;
	totdf += infodf.value;
	}
		Not plausible

tot_info v2	if (myMethods.my_scanf(fp,i,j,c.value,ft) == 0)
	//fault -- should have != 1
	{
	myMethods.my_fputs("* EOF in table *\n");
	System.exit(0);
	}
		Big tree
		
		Likely incorrect

tot_info v3	if (r.value * c.value > MAXTBL-10)
	//fault -- should have MAXTBL instead of MAXTBL-10
	{
	myMethods.my_fputs("* table too large *\n");
	System.exit(0);
	}
		Always switching
		
		Likely incorrect

tot_info v4	if (Abs(gold) < Abs(g))
	//fault: should be: if(Abs(gold) < EPS * Abs(g))
	return (Math.exp((x*(-1)) +
	(a*Math.log(x)) - LGamma(a)) * g);
		Not plausible

tot_info v7	if (pi >= 0.0) //fault: should have > instead of >= info -= (pi * Math.log(pi));	pi >= 0.0 ^ pi <= 0 .... true when pi > 0 or pi <0 Correct only if pi is guaranteed to be >=0

Table A.3: Summary of the patches generated for tcas and replace

<table>
<tr><th>Program Version</th><th>Faulty code</th><th>Patched Code and Analysis</th></tr>
<tr><td>tcas v1</td><td>result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation > ALIM()))); //fault: should have >=ALIM</td><td>Always switch
Likely incorrect</td></tr>
<tr><td>tcas v5</td><td>enabled = (High_Confidence != 0) && (Own_Tracked_Alt_Rate <= OLEV) /*&& (Cur_Vertical_Sep > MAXALTDIFF)*/; //fault: missing code</td><td>Always switch
Likely incorrect</td></tr>
<tr><td>tcas v6</td><td>return (Own_Tracked_Alt <= Other_Tracked_Alt); //fault -- should have < instead of <=</td><td>Always switch
Likely incorrect</td></tr>
<tr><td>tcas v9</td><td>if(Inhibit_Biased_Climb() >= Down_Separation) //fault: should have > instead of >=
upward_preferred = 1;</td><td>Always switch
Likely incorrect</td></tr>
<tr><td>tcas v20</td><td>if(Inhibit_Biased_Climb() >= Down_Separation) //fault: should have > instead of >=</td><td>Always switch
Likely incorrect</td></tr>
<tr><td>tcas v25</td><td>result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation > ALIM())); //fault -- should have >= instead of ></td><td>Always switch
Likely incorrect</td></tr>
<tr><td>tcas v26</td><td>enabled = (High_Confidence != 0) && /*(Own_Tracked_Alt_Rate <= OLEV) &&*/ (Cur_Vertical_Sep > MAXALTDIFF); //fault -- deleted clause</td><td>Always switch
Likely incorrect</td></tr>
<tr><td>tcas v27</td><td>enabled = (High_Confidence != 0) && (Own_Tracked_Alt_Rate <= OLEV) /*&& (Cur_Vertical_Sep > MAXALTDIFF)*/; //fault -- deleted clause</td><td>Always switch
Likely incorrect</td></tr>
<tr><td>tcas v39</td><td>result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation > ALIM())); //fault -- should have >=</td><td>Always switch
Likely incorrect</td></tr>
<tr><td>replace v7</td><td>return (c == '%'|| c == '?'/*EOL mutation BUG!*/ || c == CLOSURE); //fault: should have ‘$’ instead of ‘?’</td><td>c=’?’
Correct</td></tr>
<tr><td>replace v8</td><td>return (c == '%'|| c == '$' /*|| c == '*' */); //fault: deleted clause</td><td>C=’*’ || C=’A’
Likely incorrect</td></tr>
<tr><td>replace v16</td><td>return (c == '%'|| c == '$' || c == '*'|| c == '?'); //fault -- added " || c == '?'"</td><td>C=’?’
Correct</td></tr>
<tr><td>replace v28</td><td>return (c == '%'|| c == '$' || c == '*' || c == '['); //fault -- added " || c == '['"</td><td>C=’[‘
Correct</td></tr>
<tr><td>replace v30</td><td>return (c == '%'|| c == '$' || c == '*' || c != 'c'); //fault -- added " || c != 'c'"</td><td>C != ‘c’
Correct</td></tr>
</table>

Bibliography

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[2] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, 2002.

[3] M. Lake, "Epic failures: 11 infamous software bugs," *ComputerWorld, Sept*, 2010.

[4] J. M. Voas, "Pie: a dynamic failure-based technique," *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 717–727, 1992.

[5] C. Trad, R. Abou Assi, W. Masri, and F. Zaraket, "Cfaar: Control flow alteration to assist repair," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 208–215, 10 2018.

[6] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 295–306, 2015.

[7] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, pp. 54–72, Jan 2012.

[8] R. A. Assi and W. Masri, "Identifying failure-correlated dependence chains," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 607–616, 2011.

[9] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of 16th International Conference on Software Engineering*, pp. 191–200, 1994.

[11] M. El-Ghali and W. Masri, "Intrusion detection using signatures extracted from execution profiles," in *2009 ICSE Workshop on Software Engineering for Secure Systems*, pp. 17–24, 2009.

[12] W. Masri, R. Abou Assi, and M. El-Ghali, "Generating profile-based signatures for online intrusion and failure detection," *Information and Software Technology*, vol. 56, no. 2, pp. 238–251, 2014.

[13] R. Abou Assi, W. Masri, and C. Trad, "Substate profiling for effective test suite reduction," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 123–134, IEEE, 2018.

[14] R. Abou Assi, W. Masri, and C. Trad, "Substate profiling for enhanced fault detection and localization: An empirical study," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 16–27, IEEE, 2020.

[15] R. Just, D. Jalali, and M. D. Ernst, "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*, ISSTA 2014, (New York, NY, USA), p. 437–440, Association for Computing Machinery, 2014.

[16] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with ample," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, (New York, NY, USA), p. 99–104, Association for Computing Machinery, 2005.

[17] T. Denmat, M. Ducassé, and O. Ridoux, "Data mining and cross-checking of execution traces: A re-interpretation of jones, harrold and stasko test information," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, (New York, NY, USA), pp. 396–399, ACM, 2005.

[18] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, (New York, NY, USA), p. 467–477, Association for Computing Machinery, 2002.

[19] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *Proceedings of the ACM SIGPLAN 2003 conference*

on Programming language design and implementation, PLDI '03, (New York, NY, USA), p. 141–154, Association for Computing Machinery, 2003.

[20] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[21] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pp. 30–39, 2003.

[22] T. Xie and D. Notkin, "Checking inside the black box: regression testing by comparing value spectra," *IEEE Transactions on Software Engineering*, vol. 31, pp. 869–883, Oct 2005.

[23] R. Santelices, J. A. Jones, Yanbing Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 56–66, 2009.

[24] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," *SIGPLAN Not.*, vol. 41, no. 6, p. 169–180, 2006.

[25] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 364–374, 2009.

[26] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *Proceedings of the 6th International Workshop on Constraints in Software Testing*, CSTVA 2014, (New York, NY, USA), p. 30–39, Association for Computing Machinery, 2014.

[27] B. Elkarablieh and S. Khurshid, “Juzi: A tool for repairing complex data structures,” in *Proceedings of the 30th International Conference on Software Engineering*, ICSE ’08, (New York, NY, USA), pp. 855–858, ACM, 2008.

[28] D. Gopinath, M. Z. Malik, and S. Khurshid, “Specification-based program repair using sat,” in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 173–188, Springer Berlin Heidelberg, 2011.

[29] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 691–701, 2016.

[30] S. Mechtaev, J. Yi, and A. Roychoudhury, “Directfix: Looking for simple program repairs,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 448–458, 2015.

[31] V. Debroy and W. E. Wong, “Using mutation to automatically suggest fixes for faulty programs,” in *2010 Third International Conference on Software Testing, Verification and Validation*, pp. 65–74, 2010.

[32] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), p. 166–178, Association for Computing Machinery, 2015.

[33] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, “Precise condition synthesis for program repair,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 416–426, 2017.

[34] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, “Nopol: Automatic repair of conditional

statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[35] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," in *IEEE Transactions on Software Engineering*, vol. 38, pp. 54–72, 2012.

[36] J. Pearl, "Causal inference in statistics: An overview," *Statistics Surveys*, vol. 3, pp. 96–146, 01 2009.

[37] J. Pearl, *Causality: Models, Reasoning, and Inference.* USA: Cambridge University Press, 2000.

[38] D. B. Rubin, "Estimating causal effects of treatments in randomized and nonrandomized studies," *Journal of Educational Psychology*, vol. 66, no. 5, pp. 688–701, 1974.

[39] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, (New York, NY, USA), pp. 199–209, ACM, 2011.

[40] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, (New York, NY, USA), p. 73–84, Association for Computing Machinery, 2010.

[41] G. K. Baah, A. Podgurski, and M. J. Harrold, "Mitigating the confounding effects of program dependences for effective fault localization," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference*

on Foundations of Software Engineering, ESEC/FSE '11, (New York, NY, USA), p. 146–156, Association for Computing Machinery, 2011.

[42] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, pp. 1236–1256, Dec 2015.

[43] A. Authors, "Appendix-1 and subject programs," 10 2017.

[44] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pp. 39–46, 2006.

[45] K. Krishna and M. N. Murty, "Genetic k-means algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, no. 3, pp. 433–439, 1999.

[46] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," tech. rep., Stanford, 2006.

[47] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*, ICSE '06, (New York, NY, USA), p. 272–281, Association for Computing Machinery, 2006.

[48] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 727–739, Association for Computing Machinery, 2017.

[49] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, (New York, NY, USA), p. 702–713, Association for Computing Machinery, 2016.

[50] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), p. 441–444, Association for Computing Machinery, 2016.

[51] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, (New York, NY, USA), p. 24–36, Association for Computing Machinery, 2015.

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

[53] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 471–482, 2015.

[54] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, (New York, NY, USA), pp. 61–72, ACM, 2010.

[55] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra, "Data-guided repair of selection statements," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), p. 243–253, Association for Computing Machinery, 2014.

[56] W. Masri, R. Abou Assi, and M. El-Ghali, "Generating profile-based signatures for online intrusion and failure detection," *Information and Software Technology*, vol. 56, no. 2, pp. 238–251, 2014.

[57] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: A program query language," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), p. 365–383, Association for Computing Machinery, 2005.

[58] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European conference on object-oriented programming*, pp. 220–242, Springer, 1997.

[59] D. Lorenzoli, L. Mariani, and M. Pezze, "Towards self-protecting enterprise applications," in *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*, pp. 39–48, 2007.

[60] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[61] H.-A. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection.," in *USENIX security symposium*, vol. 286, San Diego, CA, 2004.

[62] D. Brumley, H. Wang, S. Jha, and D. Song, "Creating vulnerability signatures using weakest preconditions," in *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pp. 311–325, IEEE, 2007.

[63] D. Brumley, J. Newsome, D. Song, Hao Wang, and Somesh Jha, "Towards automatic generation of vulnerability-based signatures," in *2006 IEEE Symposium on Security and Privacy (S P'06)*, pp. 15 pp.–16, 2006.

[64] W. Lee, S. J. Stolfo, and K. W. Mok, "A data mining framework for building intrusion detection models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No. 99CB36344)*, pp. 120–132, IEEE, 1999.

[65] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.," in *NDSS*, vol. 5, pp. 3–4, Citeseer, 2005.

[66] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 116–127, 2007.

[67] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and Weibo Gong, "Anomaly detection using call stack information," in *2003 Symposium on Security and Privacy, 2003.*, pp. 62–75, 2003.

[68] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings 1996 IEEE Symposium on Security and Privacy*, pp. 120–128, 1996.

[69] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *European Symposium on Research in Computer Security*, pp. 326–343, Springer, 2003.