AMERICAN UNIVERSITY OF BEIRUT

# MODELING SOFTWARE BEHAVIOR VIA STATE PROFILING

by
RAWAD IMAD ABOU ASSI

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 Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
April 2014

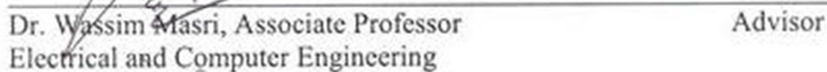
AMERICAN UNIVERSITY OF BEIRUT

# MODELING SOFTWARE BEHAVIOR VIA STATE PROFILING

by
RAWAD IMAD ABOU ASSI

Approved by:

| | |
|---|---|
| Dr. Ayman Kayssi, Professor<br>Electrical and Computer Engineering | Chair of Committee |



| | |
|---|---|
| Dr. Wassim Masri, Associate Professor<br>Electrical and Computer Engineering | Advisor |

| | |
|---|---|
| Dr. Ali El-Hajj, Professor<br>Electrical and Computer Engineering | Member of Committee |

| | |
|---|---|
| Dr. Haidar Harmanani, Professor<br>Lebanese American University | Member of Committee |

| | |
|---|---|
| Dr. Rached Zantout, Associate Professor<br>Beirut Arab University | Member of Committee |

Date of dissertation defense: April 28, 2014

## AMERICAN UNIVERSITY OF BEIRUT

## THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: ______________________________________________
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, **three years after the date of submitting my thesis, dissertation, or project,** 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.

______________________________________________

Signature Date

# ACKNOWLEDGMENTS

I'm forever grateful to my family and everybody who loved and supported me in the times when I didn't have any contribution worthy of writing acknowledgements.

I deeply thank my advisor and friend Dr. Wassim Masri for his consistent support. Ever since I met him eight years ago, I've regarded him as an ethical person and a respectable scholar. This dissertation wouldn't have been possible without his commitment, patience, and understanding.

My appreciation goes to Dr. Ayman Kayssi, Dr. Ali El-Hajj, Dr. Haidar Harmanani, and Dr. Rached Zantout for participating in the committee and for being always responsive. I'm also thankful to Dr. Fadi Zaraket who provided valuable input concerning the topic of test case intent verification.

# AN ABSTRACT OF THE DISSERTATION OF

Rawad Imad Abou Assi for Doctor of Philosophy
Major: Electrical and Computer Engineering

Title: Modeling Software Behavior via State Profiling

Many software testing and fault localization techniques rely on analyzing execution profiles which comprise runtime coverage information such as statements, branches, definition-use pairs, etc. Coverage information could be used to evaluate the quality of testing, perform test suite minimization, devise distance metrics to compare test cases, and help pinpoint the faulty code by contrasting the execution profiles of passing and failing tests. This dissertation aims at proposing and evaluating new types of execution profiles to complement existing ones especially in the cases where the runtime scenario being considered is too complex to be modeled by traditional profiles. Specifically, the proposed approaches model the runtime behavior via complex structures that involve state information, combinations of structural elements, and sequence data. In this regard, we first introduce the concept of dependence chains to assist in automated fault localization. Dependence chains capture, in addition to statement coverage, the underlying data- and control-dependence information as well as predicates describing the values of the relevant variables. We also propose using combinations of simple program elements, as opposed to individual ones, for an online intrusion detection system. Similarly, we present a mechanism for regression testing that allows users to define test requirements characterizing specific behaviors to be covered at runtime instead of choosing from a pool of pre-defined and generic program elements. Such test requirements are automatically migrated across versions and are built using three types of constructs: structural elements, first-order logic predicates specifying the state of select program variables, and sequence information.

In addition to proposing new types of execution profiles, we explore several ways to improve the effectiveness of coverage-based software analysis techniques. In particular, we present an approach that aims at identifying coincidentally-correct test cases which are shown to impair the effectiveness of coverage-based fault localization. We also address the issue of mitigating the impact of high dimensionality that's present in most types of execution profiles.

All of the proposed techniques were implemented for the java platform and the task of execution profiling was achieved by instrumenting at the bytecode level. Besides, the evaluation involved several case studies as well as empirical analysis using real subject programs with sizable test suites; these included known benchmarks, utility programs, and web servers. The results we obtained indicate that modeling program behavior via complex structures, including those that incorporate state information, is effective at capturing runtime scenarios that might go untested using traditional execution profiles. Overall, our proposed techniques were shown to have a positive impact on fault localization, regression testing, and intrusion detection.

# CONTENTS

ACKNOWLEDGEMENTS ........ v

ABSTRACT ........ vi

Chapter

I. INTRODUCTION ........ 1

A. Background ........ 1

B. Coverage-based Fault Localization ........ 2

C. Intrusion Detection ........ 3

D. Regression Testing ........ 4

E. Dimensionality Reduction ........ 5

II. FAULT LOCALIZATION USING DEPENDENCE CHAINS ........ 6

A. Related Work ........ 7

B. Suspiciousness Metric and Motivation ........ 9

C. Technique Description ........ 13

1. Background Definitions ........ 13

2. Algorithm ........ 15

3. Implementation ........ 19

D. Experimental Work ........ 20

1. Subject Programs ........ 21

2. Results ........ 21

III. MITIGATING THE IMPACT OF COINCIDENTAL CORRECTENESS ON FAULT LOCALIZATION ........ 24

A. Motivation ........ 25

B. Cleansing Techniques ........ 28

1. Basic Concepts ........ 29

2. Tech-I ........ 30

3. Tech-II ........ 31

C. Empirical Evaluation ........ 35

1. Evaluation Metrics ........ 35

2. Subject Programs ........ 37

3. Results ........ 39

4. Analysis ........ 45

D. Related Work ........ 47

IV. ONLINE INTRUSION DETECTION USING PROFILE-BASED SIGNATURES ........ 49

A. Execution Profiling ........ 52

B. Signature Generation ........ 53

C. Signature Matching ........ 58

1. Instrumentation Module ........ 59

2. Matching Module ........ 61

D. Empirical Evaluation ........ 62

1. Subject Programs and Test Suites ........ 62

2. Experimental Setup ........ 64

3. Results ........ 66

4. Profiling and Cost Analysis ........ 71

E. Related Work ........ 72

V. USER-DEFINED COVERAGE CRITERION FOR TEST CASE INTENT VERIFICATION ........ 75

A. Definitions and Notations ........ 78

B. Motivation ........ 80

C. Methodology and Implementation ........ 83

1. Specifying Test Requirements ........ 83

2. Migrating Test Requirements across Versions ........ 86

3. Checking the Coverage of Test Requirements ........ 92

D. Case Studies ........ 93

1. Testing a Bug Fix ........ 93

2. Testing Scenarios of an Algorithm ........ 96

E. Related Work ........ 100

VI. LOSSLESS REDUCTION OF EXECUTION PROFILES ........ 102

A. Motivating Example ........ 103

B. Proposed Approach ........ 105

C. Experimental Study ........ 110

1. Methodology ........ 110

2. Subject Programs and Profiling Types ........ 110

3. Metrics ........ 111

4. Results ........ 111

D. Benefits for State-based Comparison of Test Cases ........................................ 113
1. Dissimilarity Metrics ........................................ 114
2. Results ........................................ 115
VII. CONCLUSIONS AND FUTURE WORK ........................................ 117
A. Thesis Contributions ........................................ 117
B. Future Work ........................................ 119
REFERENCES ........................................ 121

# CHAPTER I

# INTRODUCTION

Dynamic software analysis encompasses a wide range of tasks that aim at ensuring the reliability of a software product. Characterizing software behavior is a valuable task in testing as it allows quantifying the quality of a test suite, comparing test cases, as well as pinpointing the faulty code when failures are observed. The general approach is to instrument the given program, run it on a given test suite, and collect execution profiles for each test case. Afterward, all the aforementioned analyses could be performed by considering the execution profiles induced by each test case. Many approaches were proposed where each uses a particular aspect of execution as a basis for profiling. In most of the cases, profiling is done to keep track of program elements being executed such as statements, branches, def-use pairs, method calls, information flows, etc. This dissertation aims at proposing and evaluating new types of execution profiles to complement existing ones especially in the cases where the bug/scenario at hand is too complex to be modeled by traditional profiles. Specifically, the proposed approaches model the runtime behavior via complex structures that involve state profiling, dependence information, combinations of structural elements, and sequence data.

## A. Background

State profiling is simply the task of recording the values assigned to the program variables at runtime. The idea of leveraging the values of variables has been first proposed

by Xie and Notkin in [1]. They collected value spectra and computed their differences to assist regression testing; their focus was on global variables and function parameters. In [2], Xie *et al* explored representing the state of objects and determining the equivalence between them. The purpose was to identify redundant unit tests from automatically generated test suites. In [3], Francis and Podgurski presented an empirical study of the effectiveness of test case filtering techniques using object-state profiling. Their profiles contained a count of how many times each possible object-state occurred during the execution of the program. They compared their results to structural profiling techniques, but their observations were not decisive. Parnin and Orso [4] argue that understanding the root causes of failures might not be achieved simply be inspecting the suspicious statement(s) provided by state-of-the-art fault localization techniques. Instead, extra activities including the inspection of program state are needed. Daikon [5] discovers certain invariants concerning selected variables. Such invariants are in the form of boolean formulae and are generated based on the values assumed by these variables in a set of runs.

In this work we propose using state profiling, in addition to other types of runtime information, to enhance software analysis. In the remaining sections we briefly describe the major areas of research that would benefit from our proposed approaches and summarize our contribution in that regard.

### B. Coverage-based Fault Localization

Coverage-based fault localization techniques aim at locating faulty code by first identifying the executing program elements that correlate the most with failure. It often

happens that the correlation measure of such elements is not high enough to successfully guide the developer to the fault. This shortcoming is likely due to the fact that the covered program elements are simple (e.g., statements, branches, or def-use pairs), and thus, cannot characterize most defects that are typically non trivial.

In chapter 2, we present a technique that identifies short dependence chains that are highly correlated with failure [6], which we term *failure-correlated dependence chains*. In addition to considering data and control dependences, we augment each chain by computing a set of predicates involving the source values and target values of its edges. This supplementary state information can potentially help identify failure-correlated chains that are shorter in length and can better assist in locating the faulty code.

Also, in chapter 3 we tackle the issue of coincidental correctness which impairs the accuracy of coverage-based fault localization [7]. A coincidentally correct test case is a one that exercises the fault and yet produces a correct output. Coincidental correctness is shown to be prevalent and it's a safety reducing factor in automated fault localization techniques because it leads to underestimating the suspiciousness of faulty program elements.

### C. Intrusion Detection

Intrusion detection systems, or *IDS*s, are categorized into two basic design approaches: anomaly-based and signature-based. The first operates by collecting information on normal or safe behavior and identifies attacks that vary from this expected set, while the second examines incoming executions for patterns of attacks that match its collection of attack signatures. While anomaly-based intrusion detection mechanisms can

detect previously un-encountered attacks, they might suffer from a high rate of false positives. On the other hand, signature-based approaches, although potentially exhibiting less false positives, are unable to detect attacks not in their collection of attack signatures and thus can result in a higher rate of false negatives. Chapter 4 investigates a signature-based technique to application-based intrusion detection [8]. The proposed technique generates signatures in the form of combinations of structural elements comprising method calls, method call pairs, basic blocks, basic block edges, and def-use pairs.

### D. Regression Testing

The goal of regression testing is to ensure that the behavior of existing code, believed correct by previous testing, is not altered by new program changes. We argue that the primary focus of regression testing should be on code associated with: a) earlier bug fixes; and b) particular application scenarios considered to be important by the tester. Existing coverage criteria do not enable such focus. For example, 100% branch coverage does not guarantee that a given bug fix is exercised or a given application scenario is tested. Therefore, there is a need for a new and complementary coverage criterion in which the user can define a test requirement characterizing a given behavior to be covered as opposed to choosing from a pool of pre-defined and generic program elements. We propose this new methodology and call it *UCov* [9], a user-defined coverage criterion wherein a test requirement is an execution pattern of program elements and predicates describing the program state. Our proposed criterion is not meant to replace existing criteria, but to complement them as it focuses the testing on important code patterns that could go untested

otherwise. *UCov* supports test case intent verification. For example, following a bug fix, the testing team may augment the regression suite with the test case that revealed the bug. However, this test case might become obsolete due to code modifications not related to the bug. But if an execution pattern characterizing the bug was defined by the user, *UCov* would determine that test case intent verification failed. This methodology is presented in chapter 5.

### E. Dimensionality Reduction

Execution profiles form the basis of many dynamic program analysis techniques developed to solve problems in fields such as test suite minimization and program comprehension. A typical profile comprises information that approximates the execution path of a program, specifically, the frequency of occurrence of program elements that are structural in nature, such as statements, branches, and def-use pairs. One major limitation is the high dimensionality present in most types of profiles, which is likely to diminish the effectiveness of several techniques based on them. Chapter 6 addresses this problem by presenting an approach that performs a lossless reduction on execution profiles [10]. It also shows how state-based comparison of test cases [11] could benefit from such reduction mechanism.

# CHAPTER II

# FAULT LOCALIZATION USING DEPENDENCE CHAINS

Coverage-based fault localization techniques generally entail two main steps. First, they identify the executing program elements that correlate most with failure. Second, starting from these elements, which are not necessarily the causes of the failure, they try to locate the faulty code following some examination strategy. It often happens that in the first step the correlation measure of the identified elements is not high enough to successfully guide the developer to the fault. This shortcoming is likely due to the fact that the program elements covered are simple, and therefore, cannot characterize most defects that are typically more complex. This calls for covering program elements whose complexity matches the complexity of the defect under consideration. Noting that a less complex element cannot characterize the defect to begin with; whereas an excessively complex element is likely to subsume the defect and successfully characterize it, but might lead to erroneously tagging too many elements as suspicious. Our ultimate goal then is to arrive at a program element that characterizes as closely as possible the defect at hand. In this chapter we present the notion of dependence chains [6], whose intermediate nodes represent (statement, variable) pairs and edges denote data or control dependence. We aim at identifying short dependence chains that are highly correlated with failure, which we term *failure-correlated dependence chains*. For more effective fault localization, both the number and length of these failure-correlated chains should be minimal. In addition to considering data and control dependences, we augment each chain by computing a set of

predicates involving the source values and target values of its edges. This supplementary state information can potentially help identify failure-correlated chains that are shorter in length and can better assist in locating the faulty code. It should be noted that our goal here is not to directly locate faults, but to identify failure-correlated program elements (namely, dependence chains) that could subsequently lead to the fault following a given examination strategy.

The rest of the chapter is organized as follows. Section A surveys related work. Section B describes the suspiciousness metric used and provides an example that motivates our work. Section C describes our technique and algorithms. Finally, Section D describes our experimental work and presents our results.

## A. Related Work

As previously mentioned, coverage-based fault localization techniques generally entail identifying failure-correlated program elements followed by locating the faulty code using some examination strategy. Our focus in this work is on the former only, that is, what program elements are better to use.

Jones et al. [12] presented a technique that uses visualization to assist with locating faults. They implemented their technique in a tool called Tarantula. The technique uses color to visually map the participation of each program statement in the outcome of the execution of the program with a test suite, consisting of both passing and failing test cases. To provide the visual mapping, the program statements are colored using a continuous spectrum from red to yellow to green: the greater the percentage of failing test

cases that execute a statement, the more red the statement should appear. A statement $s_1$ is considered more suspicious (more red) than statement $s_2$ if $M(s_1) > M(s_2)$ where

$$M(s) = \frac{\%fail(s)}{\%fail(s) + \%pass(s)}$$

In the above equation, which determines the suspiciousness or color of statement s, *%fail(s)* is the ratio of the number of failing runs that executed s to the total number of failing runs, and *%pass(s)* is the ratio of the number of passing runs that executed s to the total number of passing runs.

Denmat et al. [13] studied the limitations of the technique presented by Jones et al. [12]. They argued that for it to be effective, the following three requirements must hold: 1) a defect is due to a single faulty statement, 2) statements are independent of each other, and 3) executing a faulty statement leads most of the time to a failure. Clearly, the aforementioned requirements are not likely to be fulfilled when dealing with complex programs involving non-trivial defects. Renieris and Reiss [14] described a technique that produces a report of the "suspicious" parts of a program by analyzing the spectra differences between the faulty run and the correct run that most resembles it. The experiments they conducted used basic block coverage spectra whereas the technique proposed here involves much more complex spectra based on dependence chains. Dallmeier et al. [15] presented a tool for Java programs that locates likely failure-causing classes by comparing method call sequences of passing and failing runs. Clause and Orso [16] presented a technique for debugging failures that occur while the software runs on user platforms. Their technique allows for recording, replaying, and minimizing user executions. The resulting minimized execution can then be used to debug the defect(s)

leading to the observed failure. Santelices et al. [17] presented a Tarantula-based technique in which a statement is assigned the maximum or average of the following three scores: score computed using statement coverage, branch coverage, and def-use coverage.

### B. Suspiciousness Metric and Motivation

Let $E$ denote the type of the program element to cover, and $e$ an instance of $E$. In our experiments, presented in Section D, we employ the following suspiciousness metric that we define for statements and dependence chains:

$\boldsymbol{M_E(e) = F - P}$ where

$e$ = a dependence chain of some given length or a statement

$F = f / f_T$

$P = p / p_T$

$f$ = number of failing runs that executed $e$

$f_T$ = total number of failing runs

$p$ = number of passing runs that executed $e$

$p_T$ = total number of passing runs

$M_E$ ranges from -1.0 to 1.0 and the goal of our work is to be able to identify a small set of chains with an $M_E$ value of 1.0 based on which the faulty code will be located. We opted to use the above metric as opposed to other metrics already proposed in the literature (e.g., Tarantula), due to its simplicity and intuitiveness. For example, if $F = 0.1$ and $P = 0.0$, $M_E$ would be 0.1 (indicating a mild suspiciousness), whereas the Tarantula metric described above would be 1.0 (indicating a strong suspiciousness), which overstates the

suspiciousness of the given program element. Next we present our motivating example, and in order to make our discussion more comprehensive we will assume that an examination strategy is used following the computation of the suspiciousness metrics. We will adopt the strategy presented by Jones and Harrold [18] and used in Tarantula [12]. The strategy assumes that all statements are ranked and expects the developer to examine statements from the top of the ranking scale then down until a faulty statement is found. That is, all executed statements in a program are ranked in terms of their likelihood of being faulty by comparing the chains induced by the failing runs to the ones induced in the passing runs. The ranking of the statements associated with a given chain is determined by contrasting the percentage of failing runs to the percentage of passing runs that induced it. The statements associated with a chain are the sources and targets of its edges.

As a motivating example for covering chains (borrowed from [19]), consider the Java method shown in Table 1 where statement 5 is faulty; the + operator should have been a `-`. Note how when `(x[i] < 0)` both the faulty and correct statements assign the same value to `y` except when `x[i]` is equal to `-1`. Therefore, the failure is triggered only in the case when one or more elements of `x[]` are equal to `-1`. Table 1 also shows the following: a) Six test cases each comprising three elements of `x[]`, two of the test cases trigger a failure and the other four do not. b) The statement coverage information for each test case: a check mark indicates that the statement at the given row was executed at least once using the test case at the given column. c) The values of the suspiciousness metric computed based on statement coverage as opposed to chain coverage, i.e., in $M_E(e)$ $e$ is a statement.

**Table 1. Java code and statement coverage information for the motivating example**

| /* Statement 5 is faulty. The correct statement is: `y = -x[i] - 1/x[i]; */` <br> `public static void foo(int [] x)` <br> `{` | Passing Test Cases | | | | Failing Test Cases | | $M_E$ |
|---|---|---|---|---|---|---|---|
| | 1, 2, -3 | 0, 1, -2 | -2, -3, -4 | -5, -300, 1 | -3, -1, -100 | 100, 1, -1 | |
| 1 `int y; int z;` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| 2 `for (int i = 0; i < x.length; i++){` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| 3 `y = 0;` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| 4 `if (x[i] < 0) {` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| 5 `y = -x[i] + 1/x[i];` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| 6 `} else if (x[i] > 0) {` | ✓ | ✓ | | ✓ | | ✓ | -0.25 |
| 7 `y = x[i] - 1/x[i];` | ✓ | ✓ | | ✓ | | ✓ | -0.25 |
| `}` | | | | | | | |
| 8 `if (y == 0) {` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| **9** **`z = ...`** | ✓ | ✓ | | ✓ | ✓ | ✓ | 0.25 |
| `} else {` | | | | | | | |
| 10 `z = ...` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| `}` <br> `}` <br> 11`}` | | | | | | | |

As shown, all failing and all passing runs executed the faulty statement (statement 5) leading to an $M_E$ value of 0.0, which is clearly not high. In addition, several other statements share the same $M_E$ value and one even has a higher $M_E$ value of 0.25. Clearly, for this example, a ranking scheme based on statement coverage would not be of much help for the developer in locating the fault.

Table 2 shows how using dependence chains of length 1 is no more effective than statement coverage. In this case a chain is either a direct *control dependence* or a *def-use*. As shown, the chains involving the faulty statement are the direct control dependence *cd*(4, 5) and the def-use *du*(y, 5, 8), which are not ranked the highest given that 7 out of 11 chains are equally or higher ranked. In addition, no chain of length one is highly correlated with failure, which calls for trying a more complex program element.

**Table 2. Coverage information for chains of length 1**

| *cd(src, trgt)* *du(var, src, trgt)* | Passing Test Cases | | | | Failing Test Cases | | $M_E$ |
|---|---|---|---|---|---|---|---|
| | 1,2,-3 | 0,1,-2 | -2,-3,-4 | -5,-300,1 | -3,-1,-100 | 100,1,-1 | |
| *cd*(2,3) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| *cd*(2,4) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| *cd*(2,8) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| *cd*(4,5) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| *cd*(4,6) | ✓ | ✓ | | ✓ | | ✓ | -0.25 |
| *cd*(8,9) | ✓ | ✓ | | ✓ | ✓ | ✓ | 0.25 |
| *cd*(8,10) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| *cd*(6,7) | ✓ | ✓ | | ✓ | | ✓ | -0.25 |
| *du*(y,5,8) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| *du*(y,7,8) | ✓ | ✓ | | ✓ | | ✓ | -0.25 |
| *du*(y,3,8) | | ✓ | | | | | -0.25 |

Table 3 shows how using dependence chains of length 2 is effective at locating the faulty statement. The chain comprising *du*(y,5,8)→*cd*(8,9) covers the faulty statement and is highly correlated with failure. In this case, the developer would only need to examine

three statements to locate the faulty code, namely, statements 5, 8 and 9. In this case, a single chain exhibited a very high correlation with failure, which suggests that trying a more complex program element is not needed.

**Table 3. Coverage information for chains of length 2**

| *cd(src, trgt)*<br>*du(var, src, trgt)* | **Passing Test Cases** | | | | **Failing Test Cases** | | |
|---|---|---|---|---|---|---|---|
| | `1,2,-3` | `0,1,-2` | `-2,-3,-4` | `-5,-300,1` | `-3,-1,-100` | `100,1,-1` | $M_E$ |
| `cd(2,3)→du(y,3,8)` | | ✓ | | | | | -0.25 |
| `cd(2,4)→cd(4,5)` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| `cd(2,4)→cd(4,6)` | ✓ | ✓ | | ✓ | | ✓ | -0.25 |
| `cd(2,8)→cd(8,9)` | ✓ | ✓ | | ✓ | ✓ | ✓ | 0.25 |
| `cd(2,8)→cd(8,10)` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| `cd(4,5)→du(y,5,8)` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| `cd(4,6)→cd(6,7)` | ✓ | ✓ | | ✓ | | ✓ | -0.25 |
| `cd(6,7)→du(y,7,8)` | ✓ | ✓ | | ✓ | | ✓ | -0.25 |
| `du(y,5,8)→cd(8,9)` | | | | | ✓ | ✓ | 1.0 |
| `du(y,5,8)→cd(8,10)` | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 0.0 |
| `du(y,7,8)→cd(8,9)` | ✓ | ✓ | | ✓ | | ✓ | -0.25 |
| `du(y,7,8)→cd(8,10)` | ✓ | ✓ | | ✓ | | ✓ | -0.25 |
| `du(y,3,8)→cd(8,9)` | | ✓ | | | | | -0.25 |

## C. Technique Description

Here we provide some basic definitions, present our algorithm, then describe our implementation.

## 1. Background Definitions

*Definition 1* - A *node* is a pair *(s, v)* where *s* is a statement, *v* is a variable such that *v* is defined (assigned a value) at *s*. A node represents the source or the target of a direct data or control dependence. In the case when *s* is a conditional, *v* is irrelevant and the node is called a *predicate node*. For a node *n*, we denote by *st*(*n*) the statement associated with *n*, and by *var*(*n*) the variable associated with *n*.

*Definition 2* - The direct dependences induced at node *m* are described by the pair $(\{n_1, n_2, ..., n_k\}, m)$, denoted by ***d***, that satisfies the following:

a) The $n_i$'s are nodes.

b) One of the following is true:

- $k = 1$, $n_1$ is a predicate node, and *st(m)* is control dependent on $st(n_1)$.
- *st(m)* uses the values of $var(n_1)$, $var(n_2)$, ..., $var(n_k)$ that were last defined at $st(n_1)$, $st(n_2)$, ..., $st(n_k)$, respectively.

We also denote by *sources(d)* the set $\{n_1, n_2, ..., n_k\}$, by *target(d)* the node *m*, and by *time(d)* the timestamp indicating when *d* was exercised.

*Definition 3* - A *chain* is a sequence of nodes $(n_1, n_2, ..., n_k)$ where $k \geq 2$ and $\exists$ a set of dependences $\{d_1, d_2, ..., d_{k-1}\}$ that satisfies the following:

a) $\forall\ 1 \leq i \leq k\text{-}2$, $time(d_i) < time(d_{i+1})$

b) $\forall\ 1 \leq i \leq k\text{-}1$, $n_i \in sources(d_i)$ and $n_{i+1} = target(d_i)$

c) $\nexists$ any dependence $d$ satisfying both of the following:

- $d \notin \{d_1, d_2, ..., d_{k-1}\}$
- $\exists\ 2 \leq i \leq k$ such that $n_{i-1} \notin sources(d)$ and $n_i = target(d)$ and $time(d) > time(d_{i-1})$ and $time(d) <= time(d_{k-1})$

  In other words, there exists no dependence that broke the chain.

*Definition 4* - Given a chain $c = (n_1, n_2, ..., n_k)$, we denote by *tail(c)* the node $n_k$ and define *length(c)* to be $k$-1. $\forall\ 1 \leq i \leq k-1$, $n_i$ is said to be the predecessor of $n_{i+1}$.

*Definition 5* - A chain $e = (n_1, n_2, ..., n_k)$ is said to be an *extension* of another chain $c = (m_1, m_2, ..., m_p)$ iff $k > p$ and $\forall\ 1 \leq i \leq p$, $m_i = n_i$.

*Definition 6* - A chain $c$ is said to be *maximal* in a set of chains $S$ if and only if no extension of $c$ is contained in $S$.

## 2. *Algorithm*

The basic high level steps of our algorithm (which is shown in Figure 1) are:

a) *Specify E to represent statements*, recall that $E$ is the type of the program element to cover.

Input:
1) Sequence of direct dependences exercised under a particular test case ($d_1, d_2, ..., d_n$) sorted according to their increasing timestamp
2) A max length $L_{max}$

Output:
Set of maximal chains whose length is ≤ $L_{max}$

```
1.   output = ∅;
2.   unfinished = ∅;
3.   for i=1 to n
     {
4.      chainsToBeAdded = ∅;
5.      for j=1 to di.sources.size() {
6.             chainsToBeAdded.add(
7.             new chain(di.sources[j], di.target));
        }

8.      for k=1 to unfinished.size() {
9.             Chain ch = unfinished.get(k);
10.            if (di.hasPredicateSource()==false &&
11.                   ch.tail.isPredicate()==false &&
12.                   ch.tail.equals(di.target) &&
13.                   ch.tail.predecessor ∉ di.sources)
               {
14.                   output.add(ch);
               }
15.            else if (ch.tail ∈ d.sources)
               {
16.                   Chain extended = ch.extend(d.target);
17.                   if (length(extended) == Lmax)
18.                          output.add(extended);
                      else
19.                          chainsToBeAdded.add(extended);
               }
        }

20.     unfinished.add(chainsToBeAdded);
     }

21.  for k=1 to unfinished.size() {
22.     Chain ch = unfinished.get(k);
23.     if(ch.wasExtended() == false)
24.            output.add(ch);
     }
```

**Figure 1. Algorithm for computing dependence chains**

b) *Compute the suspiciousness metric $M_E(e)$ for all executing statements.* This step involves executing a test suite on the subject program in order to collect execution profiles describing the frequency of occurrences of each statement.

c) *Exit if the highest encountered score was 1.0*. In this case, covering chains cannot improve on covering statements.

d) *Specify E to represent dependence chains of length one*, i.e., an instance *e* of *E* is a chain *c* such that *length(c)* = 1.

e) *Compute the suspiciousness metric $M_E(e)$ for all executing chains*. This step is similar to step b.

f) *Exit if the highest encountered score was 1.0*. That is, the algorithm succeeded in identifying at least one chain that entirely correlates with failure.

g) *Increase the complexity of E.* This is done by alternating between: a) augmenting the covered dependence chain with a set of predicates involving the source values and target values of its edges, and b) increasing the length of the chain by one. That is, the sequence of covered program elements looks as follows: 1) statements, 2) chains of length one, 3) chains of length one augmented with predicates, 4) chains of length two, 5) chains of length two augmented with predicates, and so on.

h) *Exit if the complexity of E renders profile collection infeasible, otherwise go to step e.* In our experiments we exit our algorithm when profile collection for a

given chain length exceeds 24 hours. Here the algorithm is considered to have partially succeeded if it was able to improve on statement coverage.

The algorithm maintains two lists of chains: *output* and *unfinished*. The *output* list contains exercised chains that can't be extended anymore either because they reached the maximum length $L_{max}$ or because they were killed by a subsequent dependence (i.e. the tail is no longer dependent on the head). The *unfinished* list contains exercised chains that didn't reach $L_{max}$ yet and whose tail is still dependent on the head. For each direct dependences pair *d* of the sequence, lines 5-7 generate a list of chains of length one each corresponding to one of the sources of *d*. Eventually, these will be added to the *unfinished* list on line 20. Lines 10-14 check for any unfinished chain that was broken by *d* and moves it to the *output* list. Lines 15-19 identify the unfinished chains that can be extended by *d*, creates an extended copy of each, and adds the extended chains to the *unfinished* list unless their length is equal to $L_{max}$ in which case they get moved to the *output* list. After all dependences are processed, lines 21-24 move the unfinished chains that weren't extended to the *output* list.

Step 7 of our algorithm involves augmenting the covered dependence chain with a set of predicates in order to arrive at a more complex chain that is more likely to characterize complex defects. In defining these predicates we consider four types of variables, namely, *boolean*, *scalar*, *string*, and *object reference*. We also categorize these predicates into those describing the source value (*source predicates*), those describing the target value (*target predicates*), and those describing the relationship between the source and target values (*relationship predicates*). The latter type is considered only when both variables are of the

same type. Note that the predicates, listed below, are computed for each edge of a given chain:

i. *Source predicates*

- source == *True*, source == *False*; applicable when the source is a *boolean*
- source > 0, source == 0, source < 0; applicable when the source is a *scalar*
- source == *null*, source != *null* ; applicable when the source is a *string* or an *object reference*

ii. *Target predicates* (similar to above)

iii. *Relationship predicates*

- source > target, source == target, source < target; applicable when the source and target are *scalar*s
- source == target, source != target; applicable when the source and target are *boolean*s, *string*s, or *reference*s

### ***3. Implementation***

In our implementation we targeted the Java platform. The challenging part of our implementation is execution profiling, i.e, developing a profiler capable of capturing the occurrences of dependence chains of some given length, and their associated value predicates.

Our tool consists of two main components: the *Instrumenter* and the *Profiler.* The preliminary step in applying the tool is to instrument the target byte code class files using the *Instrumenter*, which was implemented using the *Byte Code Engineering Library*, *BCEL* [156]. The *Instrumenter* 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 track the occurrence of direct data and control dependences as well as the values taken by their sources and targets. This basic functionality of the *Profiler* could be extended using plugin components. For this work, we wrote two plugins. The first takes as input the direct dependences as they occur and a specified chain length, and records the induced dependence chains, i.e., this plugin implements the algorithm presented in Figure 1. The second plugin takes as input the values of the sources and targets of the direct dependences, and computes the predicates described previously.

### D. Experimental Work

Our experiments mainly aim at validating whether applying our technique on a program with a single defect would successfully identify a small set of failure-correlated dependence chains. The ultimate goal, of course, is to locate the faulty code starting from this set, but this work does not address this task. This section first describes our subject programs then tries to empirically answer our research question.

### *1. Subject Programs*

Our experiments involved 18 seeded versions that are part of the Siemens test suite [157], namely, 8 *tot_info* versions, 4 *replace* versions, and 6 *tcas* versions. These programs were manually converted to Java as part of previous work [61]. It should be noted that due to constraints on our computing resources, we had to reduce the test suite sizes of several of our programs in order to complete the profile collection process. The reduction was conducted by randomly selecting 10% of the failing tests, 10% of the passing tests that exercise the faulty code, and 10% of the passing tests that do not exercise the faulty code. Also, we only used this small subset of the Siemens programs because we excluded programs that: 1) contained faulty code that is potentially hard to manage in our experimental setup (e.g., deleted code, constant mutations, faults that span several statements, faults related to array sizes), and/or 2) yielded high suspiciousness scores when statement coverage is used. Recall from our algorithm in Section C.2 that when statement coverage performs well, using chains must not be used (see step c).

### *2. Results*

Table 4 presents the results of our experiments for all the seeded versions we used. For each version they show:

1) The results computed using statement coverage:

   a. $Max_s$: the maximum $M_E$ score attained by a statement

   b. $Fault_s$: the $M_E$ score of the faulty code

2) The results computed using chain coverage. Note that we only show the results with respect to the best encountered configuration. i.e. the simplest configuration among those which led to a maximum suspiciousness score (the full set of results could be found in [6])

    a. *Pred*: whether the dependence chains were augmented with predicates

    b. *L*: length of the dependence chains

    c. *N*: number of induced chains

    d. $Max_c$: the maximum $M_E$ score attained by a chain

    e. $Fault_c$: the maximum $M_E$ score attained by a chain traversing the faulty code

Our main concern in this work is to explore the use of dependence chains to identify program elements that are highly suspicious. Table 4 shows that:

1) In 17 versions (except for *tcas_v24*), the maximum $M_E$ score attained by covering chains is greater than the maximum $M_E$ score attained by covering statements; and this applies to chains that are relatively short (≤ 3 in most cases).

2) In 9 versions, a maximum score of 1.0 is attained. Not attaining a score of 1.0 means: a) we ran out of resources, or b) the defect cannot be characterized by a dependence chain (as it seems to be the case with some of the *tcas* versions), or c) the test suite contained coincidentally correct tests, the subject of the next chapter.

3) In 8 versions, augmenting the chains with predicates improved the maximum score.

4) In 15 versions, the most suspicious chain had a length greater than one and/or was augmented with predicates, which means that our technique is more effective than the combined coverage of def-uses and direct control dependences.

5) $Fault_c$ is greater than $Fault_s$ in all 18 versions including *tot_info_v9*, *replace_v23*, and *tcas_v24* (note that Table 4 only shows the configuration that led to a maximum value of $Max_c$). This is an indication that, when it comes to locating the fault, chain coverage is likely to perform better than statement coverage.

To summarize, our technique is effective at identifying short dependence chains that are highly correlated with failure, and augmenting chains with predicates seems to enhance the effectiveness of our technique. This improved effectiveness was observed in 17 out of 18 versions.

**Table 4. Results for tot_info, replace, and tcas**

| | | Statement Coverage | | Chain Coverage (best configuration) | | | | |
|---|---|---|---|---|---|---|---|---|
| | | $Max_s$ | $Fault_s$ | *Pred* | *L* | *N* | $Max_c$ | $Fault_c$ |
| tot_info | v4 | 0.48 | 0.47 | Yes | 4 | 71716 | 0.7 | 0.58 |
| | v5 | 0.81 | 0.33 | No | 2 | 3003 | 1 | 0.67 |
| | v7 | 0.83 | 0.28 | Yes | 1 | 1104 | 1 | 1 |
| | v9 | 0.87 | 0.34 | No | 1 | 857 | 1 | 0.34 |
| | v13 | 0.72 | 0.28 | Yes | 1 | 1111 | 1 | 1 |
| | v16 | 0.8 | 0.1 | No | 2 | 3029 | 0.95 | 0.17 |
| | v18 | 0.84 | 0.14 | Yes | 1 | 1105 | 1 | 1 |
| | v20 | 0.4 | 0.09 | Yes | 1 | 1134 | 0.67 | 0.67 |
| replace | v9 | 0.64 | 0.6 | No | 2 | 2163 | 1 | 0.71 |
| | v10 | 0.64 | 0.59 | Yes | 4 | 75389 | 0.92 | 0.84 |
| | v11 | 0.64 | 0.64 | No | 2 | 2216 | 1 | 0.74 |
| | v23 | 0.71 | 0.47 | Yes | 1 | 545 | 1 | 0.47 |
| tcas | v9 | 0.77 | 0.46 | No | 2 | 283 | 0.79 | 0.7 |
| | v20 | 0.71 | 0.46 | No | 1 | 150 | 0.75 | 0.7 |
| | v21 | 0.78 | 0.48 | Yes | 1 | 229 | 0.82 | 0.69 |
| | v22 | 0.74 | 0.48 | No | 2 | 276 | 0.91 | 0.74 |
| | v24 | 0.74 | 0.48 | No | 1 | 148 | 0.74 | 0.48 |
| | v34 | 0.3 | 0.15 | No | 2 | 297 | 1 | 1 |

# CHAPTER III

# MITIGATING THE IMPACT OF COINCIDENTAL CORRECTENESS ON FAULT LOCALIZATION

The *PIE* (Propagation-Infection-Execution) model presented in [20] emphasizes that the execution of a defect is not a sufficient condition for failure, and that the propagation of the infectious state to the output is also required. This is also reiterated in the *RIP* (Reachability-Infection-Propagation) model described in [21]. It is argued in both models that for failure to be observed the following three conditions must be met: $C_R$ = the defect was executed or reached; $C_I$ = the program has transitioned into an infectious state; and $C_P$ = the infection has propagated to the output. Coincidental correctness (CC) arises when the program produces the correct output while condition $C_R$ is met but not $C_P$. We recognize two forms of coincidental correctness, weak and strong. In weak CC, $C_R$ is met, whereas $C_I$ might or might not be met; while in strong CC, both $C_R$ and $C_I$ are met [61]. Hence, a test case that satisfies the strong form of CC also satisfies its weak form.

Coverage-based fault localization (CBFL) techniques seek to: 1) identify failure-correlated program elements using test suites in which tests are tagged as failing or passing, i.e., elements that are induced by all (or most) failing runs and not induced by any (or most) passing runs; and 2) locate the faulty code using some examination strategy [12][19].

In [73] we showed that coincidental correctness is prevalent, and demonstrated that it is a safety reducing factor for a Tarantula style CBFL [12][18]. That is, when coincidentally correct tests are present, the defect will likely be ranked as less suspicious than when they are not present. Several other researchers have also studied and pointed out

the prevalence of coincidental correctness and/or its degrading effect on fault localization [25][102][122][125]. All of this motivated us to investigate techniques to cleanse test suites from coincidentally correct tests in order to enhance CBFL. This chapter presents the work published in [7] and [73] which aims at cleansing tests suites from CC tests in order to enhance the CBFL process of identifying failure-correlated program elements.

The remainder of this chapter is organized as follows. Section A motivates the work by showing how CC has a safety reducing effect on CBFL. Sections B describes our two proposed techniques. Section C presents our empirical study and discusses the findings. Finally, Section D surveys work related to coincidental correctness.

## A. Motivation

CBFL techniques generally entail two phases: $Ph_1$) identifying failure-correlated program elements using some suspiciousness metric $M$; and $Ph_2$) locating the faulty code using some examination strategy. Typically, the techniques differ in both $M$ and the examination strategy; and the performance of a given technique is attributed to both phases. This work does not make any attempts to improve $Ph_2$, but instead focuses on identifying CC tests and consequently improving $Ph_1$.

We now demonstrate how weak coincidental correctness has a safety reducing effect on CBFL. Specifically, we show that the presence of CC tests leads to $M$ values that underestimate the suspiciousness of faulty program elements, and thus identifying or removing CC tests from tests suites improves $Ph_1$. We carry this out in the context of

widely used suspiciousness metrics [101][102], namely, Jaccard [106], Tarantula [12], AMPLE [15], and Ochiai [101][102]. These metrics use the following components:

*e = faulty program element*

$a_{11}(e)$ = *# of failing runs that executed e*

$a_{01}(e)$ = *# of failing runs that did not execute e*

$a_{10}(e)$ = *# of passing runs that executed e*

$a_{00}(e)$ = *# of passing runs that did not execute e*

The *Jaccard* metric is defined as follows:

$$M(e) = \frac{a_{11}(e)}{a_{11}(e) + a_{01}(e) + a_{10}(e)}$$

Assume that *n* tests executed *e* but did not induce a failure, i.e., the test suite contains *n* CC tests. In this case, the value of *M*(e) is misleading and to arrive at a more faithful value we should subtract *n* from $a_{10}(e)$, the new value would become:

$$M'(e) = \frac{a_{11}(e)}{a_{11}(e) + a_{01}(e) + a_{10}(e) - n}$$

It is clear that $M'(e) \geq M(e)$, i.e., *not accounting for n would underestimate the suspiciousness of e*.

The main *Tarantula* suspiciousness metric is defined as follows:

$$M(e) = \frac{F(e)}{F(e) + P(e)}$$

where

$f_T$ = *total # of failing runs* = $a_{11}(e) + a_{01}(e)$

$p_T$ = total # of passing runs = $a_{10}(e) + a_{00}(e)$

$F(e) = a_{11}(e) / f_T$

$P(e) = a_{10}(e) / p_T$

In case of $n$ CC tests, the more accurate metric would then be

$$M'(e) = \frac{F(e)}{F(e) + P'(e)}$$

where $P'(e) = (a_{10}(e) - n) / (p_T - n)$. It could be easily shown that $M'(e) \geq M(e)$. To verify, $M'(e) \geq M(e) \Rightarrow 1/M'(e) \leq 1/M(e) \Rightarrow P'(e)/F(e) \leq P(e)/F(e) \Rightarrow P'(e)/P(e) \leq 1$, which holds since $a_{10}(e) \leq p_T$.

The *AMPLE* metric is:

$$M(e) = | F(e) - P(e) |$$

A more accurate value would be $M'(e) = | F(e) - P'(e) |$ where $P'(e) = (a_{10}(e) - n) / (p_T - n)$. Here also, $P'(e) \leq P(e)$, and consequently $M'(e) \geq M(e)$.

Finally, the *Ochiai* metric is defined as follows:

$$M(e) = \frac{a_{11}(e)}{\sqrt{\left(a_{11}(e) + a_{01}(e)\right) \times \left(a_{11}(e) + a_{10}(e)\right)}}$$

To arrive at a more faithful suspiciousness value we should subtract $n$ from $a_{10}(e)$ leading to

$$M'(e) = \frac{a_{11}(e)}{\sqrt{\left(a_{11}(e) + a_{01}(e)\right) \times \left(a_{11}(e) + a_{10}(e) - n\right)}}$$

which is clearly larger than $M(e)$.

It should be noted that in our empirical study (Section C) we adopt the *Ochiai* metric since it was shown to outperform the other metrics [99].

To summarize, we have shown above that cleansing test suites from CC would improve $Ph_1$ by assigning the faulty code higher (or equal) suspiciousness values. The scope of this work is to devise techniques that achieve this purpose.

### B. Cleansing Techniques

Figure 2 depicts a test suite T with its various components. It comprises a set of passing tests $T_P$ and a set of failing tests $T_F$, where $T_P$ might be composed of a subset of coincidentally correct tests $T_{CC}$ and another subset of true passing tests $T_{trueP}$. As noted earlier, $T_{CC}$ refers to either weak or strong coincidentally correct tests, depending on the context. Our aim is to identify $T_{CC}$ given $T_F$ and $T_P$ so that the tests in $T_{CC}$ would be discarded from T in order to enhance the safety of CBFL. A passing test identified by our techniques as a potential CC test will be called a $cc_t$; and the set of identified $cc_t$'s, our estimate of $T_{CC}$, will be called $T_{CC}$'. We present two techniques to achieve our goal, namely, *Tech-I* and *Tech-II*. Both techniques are based on analyzing execution profiles, which we describe next.

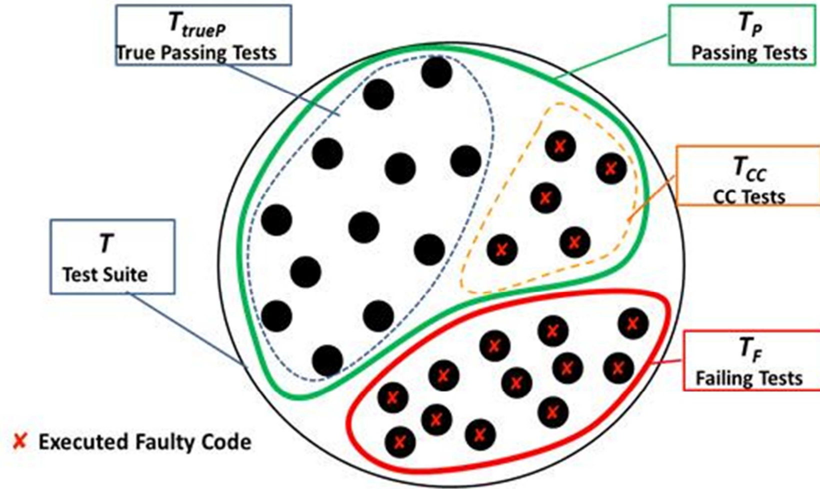



**Figure 2. A test suite with its various components**

### *1. Basic Concepts*

a. <u>Execution Profiles</u>

The execution profiles we consider consist of the following three types of program elements:

- Basic blocks (*BB*): For every basic block *B* such that *B* is executed in at least one test, a *BB* profile contains a flag indicating whether *B* is executed in the current test.
- Basic-block edges (*BBE*): For every pair of basic blocks *B1* and *B2* such that there is a branch from *B1* to *B2* in at least one test, a *BBE* profile contains a flag indicating whether this branch is taken in the current test.
- Def-use pairs (*DUP*): For every pair consisting of a variable definition *D*(*x*) and a use *U*(*x*) such that *D*(*x*) dynamically reaches *U*(*x*) in at least one test, a *DUP* profile contains a flag indicating whether *D*(*x*) dynamically reaches *U*(*x*) in the current test.

- Combined *BB*, *BBE* and *DUP* (*ALL*): Combines profiling information of *BB, BBE* and *DUP*. *ALL* profiles are likely to characterize the behavior of the target application better than each of the other types considered individually.

Our experiments in Section C involve both Java and C programs. We generate *ALL* execution profiles for the Java subjects using a tool that was developed in [67]. Whereas we generate (only) *BB* execution profiles for the C subjects using *gcov* [155].

b. Definition of $cc_e$

Given a program element *e*, we denote by *F(e)* the ratio of failing test cases executing *e*, and by *P(e)* the ratio of passing test cases executing *e*. Each test case $t_i$ is associated with a characteristic function $f_i$ defined as follows:

$$f_i(e) = \begin{cases} 1 \text{ if } e \text{ is exercised by } t_i \\ 0 \text{ otherwise} \end{cases}$$

Given a test suite *T* that exercises elements $e_1, e_2, \ldots, e_n$, a test case $t_i$ in *T* is represented by the feature vector $V_i = [f_i(e_1) \quad f_i(e_2) \quad \ldots \quad f_i(e_n)]$. Both of our techniques assume that: 1) there exists a set of elements, which we call $cc_e$'s, that correlate with coincidental correctness, and 2) a good candidate for a $cc_e$ is any program element that occurs in all failing runs and in a non-zero but not excessively large percentage of passing runs.

### 2. *Tech-I*

*Tech-I* conjectures that coincidentally correct tests are similar to failing tests in terms of their execution profiles; and hence are expected to automatically cluster together.

Our approach is to use cluster analysis [64][123], and specifically *k-means* clustering, to partition the whole test suite into two clusters, pick the cluster containing the majority of failing tests and label all passing tests within it as $cc_t$'s. We use the Euclidean metric as a distance measure and discard elements that are not $cc_e$'s. In fact, since *Tech-I* considers an element $e$ to be a $cc_e$ *iff* $F(e)=1$ and $0 < P(e) < 1$, it follows that all failing runs would collapse to the same point in the clustering process. The distance between two tests $t_i$ and $t_j$ is defined as follows:

$$d(t_i, t_j) = \sqrt{\sum_{\substack{k=1 \\ e_k \text{ is a cce}}}^{n} (f_i(e_k) - f_j(e_k))^2}$$

The two initial means consist of the common failing point and the most distant passing test.

### 3. *Tech-II*

*Tech-I* partitions the whole test suite into two clusters, expecting that one will mainly contain true passing tests, and the other failing and CC tests. On the other hand, *Tech-II* partitions only the passing tests into two clusters, expecting that one of the clusters will be comprised of only (or mostly) CC tests. *Tech-II* is an improvement of the cleansing technique presented in [73], which considers an element $e$ to be a $cc_e$ *iff* $F(e)=1$ and $0 < P(e) \leq \theta$ where $\theta$ is a non-zero threshold specified by the user. It assigns each $cc_e$ a weight equal to its Tarantula suspiciousness score and clusters the passing tests into two groups according to the $cc_e$'s they induce. The cluster associated with the larger average $cc_e$ weight is selected as the one containing the CC tests. We recognize three main limitations in that

approach that *Tech-II* tries to address. First, the user is required to specify a value for $\theta$, which restricts the automation of the technique. Second, and most importantly, we found that there is no unique value of $\theta$ that works best for all considered applications. Finally, the experiments we conducted exhibited a high rate of false positives and in some cases, a high rate of false negatives. To overcome these limitations, we modeled the $cc_e$'s as a fuzzy set. Fuzzy set theory [128] was developed to model the vagueness/subjectivity associated with certain concepts when designing intelligent systems (e.g. "high" temperature, "normal" speed, etc). Contrary to classical set theory, membership in a fuzzy set can be partial and is associated with a function that maps the elements considered into the interval [0, 1]. The membership value represents the degree of compatibility with the property defining the fuzzy set. An element with membership value of 1 is a complete member of the fuzzy set whereas an element with membership value of 0 is not a member at all. In our context, there will be no strict boundary between $cc_e$'s and non-$cc_e$'s; the transition will be gradual. Figure 3 shows function $\mu$ that defines such a fuzzy set. Our choice is guided by the empirical study we conducted in [61] and by our intention to use a generically shaped function. The value 56% represents the average of weak CC's per test suite among the 148 subject programs we used in that study. Therefore, a $cc_e$ would be given a (full) membership of 1 if it is exercised by 56% of the passing runs. Moreover, since an element executed by no passing runs can't be a $cc_e$ we set $\mu(0\%)=0$. Similarly, we set $\mu(100\%)=0$ because an element exercised by all passing runs cannot be a $cc_e$ as well.

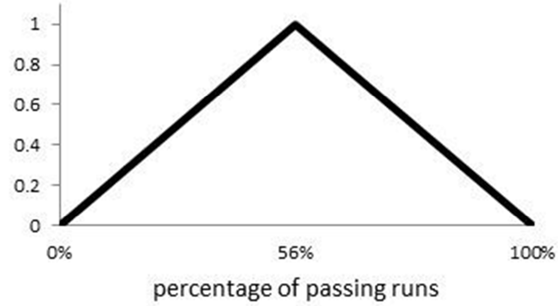



**Figure 3. The fuzzy set of cce's**

As such, CCE will be a fuzzy set defined by the following membership function:

$$f_{CCE}(\mathrm{e}) = \begin{cases} 0 \text{ if } F(\mathrm{e}) < 1 \\ \mu\big(P(\mathrm{e})\big) \text{ otherwise} \end{cases}$$

Given a set of passing test cases *S* that induces *n* program elements, we quantify the likelihood of *S* to contain CC tests using the following metric:

$$Relevance(S) = \frac{\sum_{t_i \in S} \sum_{k=1}^{n} f_i(e_k) \times f_{CCE}(e_k)}{|S|}$$

$Relevance(S_1) > Relevance(S_2)$ means that compared to $S_2$, the tests in $S_1$ induce program elements that cumulatively are more likely to be $cc_e$'s.

*Tech-II* iteratively uses *k-means* to split the passing tests into two clusters and selects the cluster having a higher relevance value as the one containing the $cc_t$'s. It stops when the relevance of the selected cluster drops below 75% of the relevance of the one chosen initially. The used distance metric incorporates the CCE membership values as follows:

$$d(t_i, t_j) = \sqrt{\sum_{k=1}^{n}\left[f_{CCE}(e_k) \times (f_i(e_k) - f_j(e_k))\right]^2}$$

At each iteration, the initial means are chosen as the two test cases separated by the largest measured distance. What follows is the pseudocode for *Tech-II*:

```
1. TCC' ← Ø
2. TP' ← TP
3. (cluster1, cluster2) ← Kmeans(TP', 2)
4. R0 ← max(relevance(cluster1), relevance(cluster2))
5. R ← R0
6. While R/R0 >= 0.75
7.     selected ← SelectClusterWithMaxRelevance(cluster1,cluster2)
8.     TCC' ← TCC' ∪ selected
9.     TP'  ← TP' - selected
10.    (cluster1, cluster2) ← Kmeans(TP', 2)
11.    R ← max(relevance(cluster1), relevance(cluster2))
```

Line 1 initializes our estimate of the coincidentally correct tests, $T_{CC}$', to the empty set. Line 2 initializes the set of passing tests that need to be analyzed, $T_P$', to $T_P$. Lines 3 and 4 create two clusters out of $T_P$', compute their respective relevance then store their maximum in $R_0$ to be used in Line 6 as part of the stopping criterion. Lines 7 through 9 select the cluster with the higher relevance, add its associated tests to $T_{CC}$', and subtract them from $T_P$'. Lines 10 and 11 create two clusters out of the now reduced $T_P$', compute their respective relevance, and stores their maximum in R. Line 6 implements the stopping criterion, i.e., the loop exists when R drops below (0.75 * $R_0$).

## C. Empirical Evaluation

### *1. Evaluation Metrics*

In order to empirically evaluate the effectiveness of our techniques we compute metrics to quantify the generated false negatives and false positives. We also compute two metrics *FaultScore* and *MaxScore* that will help assess the potential impact of our CC cleansing techniques on the safety of CBFL techniques.

a. Measure of generated false negatives and false positives

$$FN = \frac{|T_{CC} - Tcc'|}{|T_{CC}|}$$

The *FN* measure above assesses whether or not we are successfully identifying all of the coincidentally correct tests. $T_{CC}$ is the set of (true) coincidentally correct tests determined using the oracles, and $T_{CC}$' is the estimate of $T_{CC}$ computed using our techniques. In the example of Figure 4 the value of *FN* is 2/5.

$$FP = \frac{|(T_P - T_{CC}) \cap Tcc'|}{|T_P - T_{CC}|}$$

The *FP* measure assesses whether we are erroneously categorizing tests as coincidentally correct. In Figure 4, the value of *FP* is 1/12.

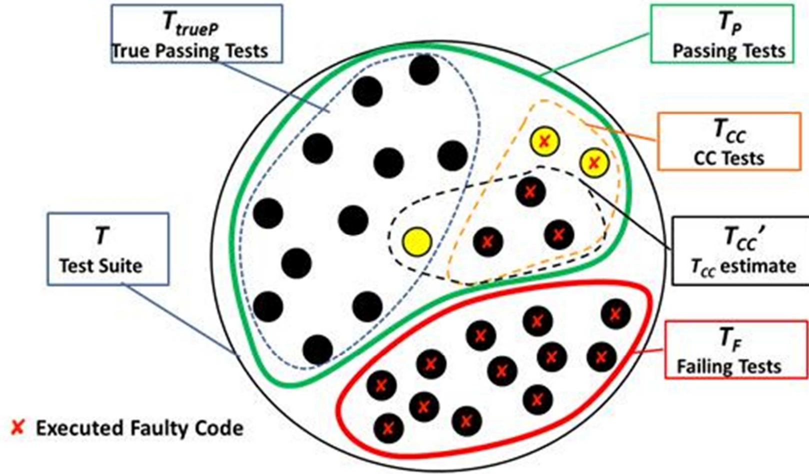



**Figure 4. TCC' estimate resulting in two false negatives and one false positive**

b. FaultScore and MaxScore

Using the *Ochiai* suspiciousness metric, *FaultScore* is the value assigned to the fault, and *MaxScore* is the maximum observed value assigned to any program element. These metrics are used as follows:

a) *MaxScore* can be used to decide whether or not to apply our CC cleansing approach to a given test suite *T*. Assuming that the used profile types are suitable enough to characterize the fault at hand, a *MaxScore* value that is strictly less than 1.0 implies that *T* contains coincidentally correct tests, and therefore, our approach is likely to be beneficial. Note that a *MaxScore* of 1.0 does not always mean that no coincidentally correct tests exist, since it is possible that elements unrelated to the defective code would have an $M(e) = 1.0$.

b) In our experiments we will gauge the change of *FaultScore* from when *T* is used to when $T–T_{CC'}$ is used. An increased value of *FaultScore* is an indication that our

cleansing techniques will lead to an improved safety of the CBFL techniques. Also, it is desirable that, after cleansing, *FaultScore* is 1.0 or at least close to 1.0.

Finally, as stated before, the scope of this work is to devise techniques for cleansing test suites from CC as this would improve the safety of CBFL; therefore, our experiments will not involve CBFL precision related tasks such as examination strategies.

### *2. Subject Programs*

Our empirical study involved the following 15 subject programs:

1) All seven of the Siemens programs [157] that were converted from C to Java as part of previous work [61].
2) Release 1, 3, and 5 of *NanoXML* [157]
3) The *JTidy HTML* and *XML* syntax checker and pretty printer *release 3.0* [158]
4) *Space*, an interpreter for an array definition language [157]
5) Three Unix utilities: the stream editor *sed*, the lexical analyzer generator *flex*, and the file compression/decompression tool *gzip*. All three were downloaded with their test suites from [157]

Table 5 provides more information about these subject programs and associated versions. Note how *Space, JTidy, sed, flex, and gzip* are relatively large and some contain real faults [105]. Our experiments involved *BB* and *ALL* execution profiles for the Java programs and only *BB* for the C programs. For the C programs, we used the gcov tool [155] to generate the profiles and we had to discard all versions that caused compile errors with the gcc version we used (*v4.3.4*). We also discarded all test cases that caused segmentation

faults (in the C programs) and those which caused exceptions being thrown (in the Java programs). Among the resulting versions/test suites, we additionally discarded the ones containing no failures, those with very few passing runs, and those for which no proper fault oracle could be used (e.g. constant mutation in header file). Table 5 summarizes the versions' information by showing for each subject program the original number of versions available (All), the number used in our evaluation (Used), the number of versions that exhibited no CCs (noCCs), and the number exhibiting no true passing tests (noTrueP). Thus, there are 363 original versions, from which only 142 are used. Also, 37 out of the 142 exhibited either no CCs or no true passing tests, and had to be evaluated separately.

**Table 5. Subject programs**

| Program | #Versions | | | | Platform | LOC | Profiling Type | Fault Type |
|---|---|---|---|---|---|---|---|---|
| | All | Used | noCCs | noTrueP | | | | |
| *print_tokens* | 7 | 4 | 1 | 0 | *Java* | 536 | *BB - ALL* | *Seeded* |
| *print_tokens2* | 10 | 8 | 2 | 0 | *Java* | 387 | *BB - ALL* | *Seeded* |
| *Replace* | 32 | 6 | 0 | 0 | *Java* | 554 | *BB - ALL* | *Seeded* |
| *Schedule* | 9 | 4 | 0 | 0 | *Java* | 425 | *BB - ALL* | *Seeded* |
| *schedule2* | 10 | 8 | 0 | 0 | *Java* | 766 | *BB - ALL* | *seeded* |
| *Tcas* | 41 | 19 | 0 | 0 | *Java* | 136 | *BB - ALL* | *Seeded* |
| *tot_info* | 23 | 17 | 1 | 0 | *Java* | 494 | *BB - ALL* | *Seeded* |
| *NanoXML r1* | 5 | 3 | 0 | 0 | *Java* | 4,334 | *BB - ALL* | *Real* |
| *NanoXML r3* | 5 | 5 | 2 | 0 | *Java* | 7,185 | *BB - ALL* | *Real* |
| *NanoXML r5* | 6 | 4 | 0 | 0 | *Java* | 7,646 | *BB - ALL* | *Real* |
| *JTidy* | 5 | 1 | 0 | 0 | *Java* | 9,153 | *BB - ALL* | *Real* |
| *Space* | 38 | 27 | 3 | 1 | *C* | 6,445 | *BB* | *Real* |
| *Gzip* | 59 | 5 | 4 | 0 | *C* | 9,251 | *BB* | *seeded* |
| *Sed* | 32 | 10 | 3 | 0 | *C* | 11,699 | *BB* | *Real-seeded* |
| *Flex* | 81 | 21 | 15 | 5 | *C* | 15,895 | *BB* | *seeded* |

### *3. Results*

Table 6 provides the following information about the 105 versions exhibiting failures, CCs, and true passing tests: a) test suite size, b) number of failing tests, c) number of weak CC tests, and d) *FaultScore* values computed using the original test suite, after applying *Tech-I*, and after applying *Tech-II* respectively. Note that since the test suites of *Space* and *JTidy* are relatively large (13,525 and 16,694 respectively), we opted to randomly select a smaller subset of 1000 test cases while preserving the same ratio of failing/CC/true passing. Also note that the fault types ranged between added/deleted code, altered conditionals, wrong assignment statements, wrong function arguments, missing conditionals, and others.

**Table 6. Versions used and results**

| Program | \|T\| | $\|T_F\|$ | \|TCC\| | *FaultScore* | | |
|---|---|---|---|---|---|---|
| | | | | *org.* | *Tech-I* | *Tech-II* |
| print_tokens_v2 | 4070 | 48 | 1546 | 0.17 | 1.00 | 0.74 |
| print_tokens_v5 | 4070 | 150 | 1250 | 0.33 | 0.63 | 1.00 |
| print_tokens_v7 | 4070 | 28 | 357 | 0.27 | 1.00 | 1.00 |
| print_tokens2_v1 | 4055 | 240 | 2841 | 0.28 | 1.00 | 0.78 |
| print_tokens2_v3 | 4055 | 33 | 726 | 0.21 | 1.00 | 0.45 |
| print_tokens2_v4 | 4055 | 332 | 1099 | 0.48 | 1.00 | 1.00 |
| print_tokens2_v7 | 4055 | 207 | 1222 | 0.38 | 1.00 | 1.00 |
| print_tokens2_v8 | 4055 | 256 | 3247 | 0.27 | 1.00 | 0.56 |
| print_tokens2_v9 | 4055 | 56 | 1373 | 0.20 | 1.00 | 1.00 |
| replace_v2 | 2843 | 11 | 1325 | 0.09 | 1.00 | 1.00 |
| replace_v7 | 2843 | 54 | 765 | 0.26 | 1.00 | 1.00 |
| replace_v8 | 2843 | 212 | 607 | 0.51 | 1.00 | 0.95 |
| replace_v16 | 2843 | 54 | 765 | 0.26 | 1.00 | 1.00 |
| replace_v28 | 2843 | 18 | 801 | 0.15 | 0.20 | 0.20 |
| replace_v30 | 2843 | 469 | 350 | 0.76 | 1.00 | 0.95 |
| schedule_v2 | 2650 | 210 | 1382 | 0.36 | 1.00 | 1.00 |
| schedule_v3 | 2650 | 159 | 1199 | 0.34 | 1.00 | 1.00 |
| schedule_v4 | 2650 | 294 | 1481 | 0.41 | 1.00 | 1.00 |
| schedule_v8 | 2650 | 31 | 1311 | 0.15 | 1.00 | 1.00 |

| schedule2_v1 | 2710 | 65 | 1802 | 0.19 | 1.00 | 0.94 |
|---|---|---|---|---|---|---|
| schedule2_v2 | 2710 | 31 | 2636 | 0.11 | 0.51 | 0.26 |
| schedule2_v3 | 2710 | 34 | 2633 | 0.11 | 0.52 | 0.26 |
| schedule2_v4 | 2710 | 2 | 2578 | 0.03 | 1.00 | 0.09 |
| schedule2_v5 | 2710 | 32 | 2628 | 0.11 | 0.52 | 0.22 |
| schedule2_v6 | 2710 | 7 | 2573 | 0.05 | 1.00 | 1.00 |
| schedule2_v7 | 2710 | 31 | 2636 | 0.11 | 0.51 | 0.26 |
| schedule2_v10 | 2710 | 46 | 2614 | 0.13 | 0.59 | 0.30 |
| tcas_v1 | 1597 | 131 | 345 | 0.52 | 1.00 | 1.00 |
| tcas_v2 | 1597 | 67 | 808 | 0.28 | 1.00 | 1.00 |
| tcas_v5 | 1597 | 10 | 1557 | 0.08 | 1.00 | 0.13 |
| tcas_v6 | 1597 | 12 | 584 | 0.14 | 1.00 | 1.00 |
| tcas_v7 | 1597 | 36 | 1531 | 0.15 | 1.00 | 0.22 |
| tcas_v9 | 1597 | 7 | 868 | 0.09 | 1.00 | 1.00 |
| tcas_v16 | 1597 | 70 | 1497 | 0.21 | 0.30 | 0.30 |
| tcas_v17 | 1597 | 35 | 1532 | 0.15 | 1.00 | 0.22 |
| tcas_v18 | 1597 | 29 | 1538 | 0.14 | 1.00 | 0.20 |
| tcas_v19 | 1597 | 19 | 1548 | 0.11 | 1.00 | 0.16 |
| tcas_v20 | 1597 | 18 | 857 | 0.14 | 1.00 | 1.00 |
| tcas_v22 | 1597 | 11 | 864 | 0.11 | 1.00 | 1.00 |
| tcas_v24 | 1597 | 7 | 868 | 0.09 | 1.00 | 1.00 |
| tcas_v25 | 1597 | 3 | 396 | 0.09 | 1.00 | 1.00 |
| tcas_v26 | 1597 | 11 | 1556 | 0.08 | 0.13 | 0.13 |
| tcas_v27 | 1597 | 10 | 1557 | 0.08 | 1.00 | 0.13 |
| tcas_v29 | 1597 | 18 | 857 | 0.14 | 1.00 | 1.00 |
| tcas_v37 | 1597 | 92 | 464 | 0.41 | 1.00 | 1.00 |
| tcas_v39 | 1597 | 3 | 396 | 0.09 | 1.00 | 1.00 |
| tot_info_v2 | 1052 | 10 | 953 | 0.10 | 0.85 | 0.34 |
| tot_info_v3 | 1052 | 3 | 1046 | 0.05 | 1.00 | 0.12 |
| tot_info_v4 | 1052 | 33 | 635 | 0.22 | 1.00 | 1.00 |
| tot_info_v5 | 1052 | 29 | 739 | 0.19 | 1.00 | 1.00 |
| tot_info_v7 | 1052 | 123 | 674 | 0.39 | 1.00 | 1.00 |
| tot_info_v8 | 1052 | 199 | 29 | 0.93 | 1.00 | 1.00 |
| tot_info_v9 | 1052 | 37 | 731 | 0.22 | 1.00 | 1.00 |
| tot_info_v11 | 1052 | 199 | 29 | 0.93 | 1.00 | 1.00 |
| tot_info_v12 | 1052 | 33 | 696 | 0.21 | 1.00 | 1.00 |
| tot_info_v13 | 1052 | 128 | 669 | 0.40 | 1.00 | 1.00 |
| tot_info_v14 | 1052 | 2 | 1047 | 0.04 | 1.00 | 0.10 |
| tot_info_v15 | 1052 | 199 | 29 | 0.93 | 1.00 | 1.00 |
| tot_info_v16 | 1052 | 170 | 793 | 0.42 | 0.96 | 0.71 |
| tot_info_v17 | 1052 | 44 | 624 | 0.26 | 1.00 | 1.00 |
| tot_info_v22 | 1052 | 23 | 843 | 0.16 | 1.00 | 0.36 |
| tot_info_v23 | 1052 | 71 | 597 | 0.33 | 1.00 | 1.00 |
| nano1_v1 | 169 | 27 | 137 | 0.41 | 0.74 | 0.55 |
| nano1_v3 | 169 | 45 | 12 | 0.89 | 1.00 | 1.00 |
| nano1_v5 | 169 | 29 | 113 | 0.45 | 1.00 | 0.84 |
| nano3_v1 | 141 | 10 | 54 | 0.40 | 0.95 | 0.85 |
| nano3_v3 | 141 | 10 | 123 | 0.27 | 0.88 | 0.38 |
| nano3_v4 | 141 | 4 | 3 | 0.76 | 1.00 | 1.00 |
| nano5_v2 | 141 | 30 | 40 | 0.65 | 1.00 | 0.97 |
| nano5_v4 | 141 | 40 | 48 | 0.67 | 1.00 | 1.00 |
| nano5_v5 | 141 | 30 | 40 | 0.65 | 1.00 | 0.97 |
| nano5_v6 | 141 | 30 | 103 | 0.47 | 0.73 | 0.63 |
| jtidy_v1 | 1000 | 10 | 863 | 0.11 | 1.00 | 0.24 |
| space_v3 | 1000 | 47 | 908 | 0.22 | 1.00 | 0.26 |

| space_v5 | 1000 | 291 | 10 | 0.98 | 1.00 | 1.00 |
|---|---|---|---|---|---|---|
| space_v6 | 1000 | 935 | 10 | 0.99 | 1.00 | 1.00 |
| space_v7 | 1000 | 11 | 18 | 0.62 | 1.00 | 1.00 |
| space_v8 | 1000 | 10 | 13 | 0.66 | 1.00 | 1.00 |
| space_v9 | 1000 | 305 | 128 | 0.84 | 1.00 | 1.00 |
| space_v10 | 1000 | 87 | 59 | 0.77 | 1.00 | 1.00 |
| space_v11 | 1000 | 69 | 95 | 0.65 | 1.00 | 1.00 |
| space_v12 | 1000 | 10 | 10 | 0.71 | 1.00 | 1.00 |
| space_v13 | 1000 | 56 | 10 | 0.92 | 1.00 | 1.00 |
| space_v14 | 1000 | 129 | 631 | 0.41 | 1.00 | 1.00 |
| space_v15 | 1000 | 254 | 413 | 0.62 | 1.00 | 1.00 |
| space_v16 | 1000 | 36 | 10 | 0.88 | 1.00 | 1.00 |
| space_v17 | 1000 | 14 | 502 | 0.16 | 1.00 | 1.00 |
| space_v18 | 1000 | 10 | 10 | 0.71 | 1.00 | 1.00 |
| space_v19 | 1000 | 90 | 10 | 0.95 | 1.00 | 1.00 |
| space_v20 | 1000 | 15 | 10 | 0.77 | 1.00 | 1.00 |
| space_v21 | 1000 | 15 | 10 | 0.77 | 1.00 | 1.00 |
| space_v23 | 1000 | 20 | 23 | 0.68 | 1.00 | 1.00 |
| space_v24 | 1000 | 52 | 229 | 0.43 | 1.00 | 1.00 |
| space_v28 | 1000 | 490 | 302 | 0.79 | 0.95 | 0.98 |
| space_v31 | 1000 | 119 | 54 | 0.83 | 1.00 | 0.88 |
| space_v33 | 1000 | 7 | 20 | 0.51 | 1.00 | 1.00 |
| flex4_v6 | 532 | 148 | 96 | 0.78 | 1.00 | 1.00 |
| gzip5_v1 | 195 | 6 | 2 | 0.87 | 1.00 | 1.00 |
| sed2_v1 | 212 | 38 | 6 | 0.93 | 1.00 | 1.00 |
| sed2_v2 | 212 | 12 | 6 | 0.82 | 1.00 | 1.00 |
| sed2_v3 | 212 | 6 | 77 | 0.27 | 1.00 | 1.00 |
| sed3_v1 | 213 | 3 | 12 | 0.45 | 1.00 | 1.00 |
| sed3_v3 | 213 | 13 | 2 | 0.93 | 1.00 | 1.00 |
| sed3_v4 | 213 | 29 | 4 | 0.94 | 0.98 | 1.00 |
| sed4_v3 | 213 | 1 | 1 | 0.71 | 1.00 | 1.00 |

Figures 5 and 6 involve all of the 105 Java and C versions listed in Table 6. They show the results of our techniques computed using *BB* execution profiles. Whereas Figures 7 and 8 show the results involving only 73 Java versions using *ALL* execution profiles. The results associated with the remaining 37 versions that exhibited no CCs or no true passing tests are presented later in this section.

The plot in Figure 5 shows the results of applying *Tech-I* to identify weak CC tests, namely, FN, FP, and (FN+FP). For clarity, the horizontal axis shows the versions

identifiers sorted in ascending order based on their respective values of (FN+FP). The following could be observed:

a. 18 versions did not exhibit any false negatives or false positives, i.e., FN and FP are both 0
b. 14 other versions exhibited a low FN+FP of less than 20%
c. The remaining versions exhibited a relatively high FN+FP mostly due to FP
d. With the exception of versions 103, 104, and 105 (and versions 1-18) a high FP was counteracted with a low FN, and vice-versa

Also, not shown in the figure, the average FN is 3% and the average FP is 45%

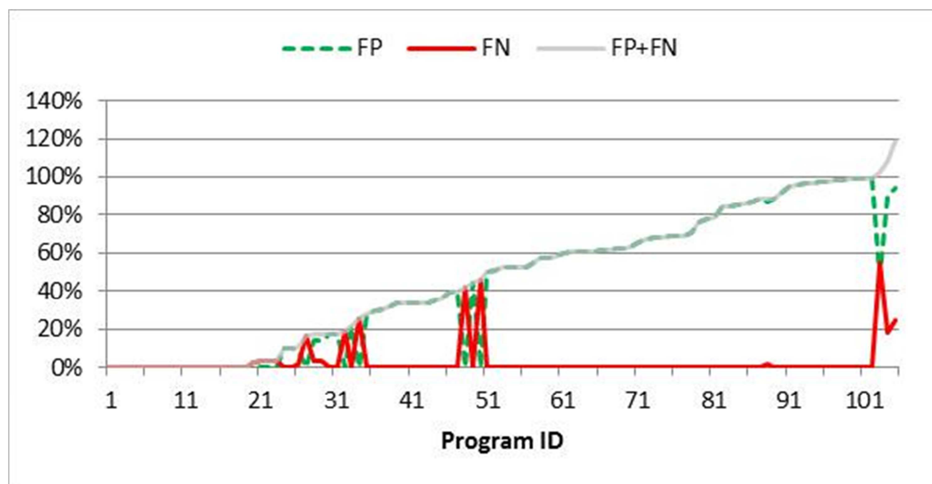



**Figure 5. CC cleansing results using *Tech-I* (BB)**

Figure 6 is the counterpart of Figure 5 using *Tech-II*. We make the following observations:

a. 15 versions (14.3%) exhibited an FN+FP of 0%
b. 21 other versions exhibited a low FN+FP of less than 20%

c. The remaining versions exhibited a relatively high FN+FP due to either FP or FN

d. With the exception of versions 52, 57, 72, 100-105 (and versions 1-15), and similarly to *Tech-I*, a high FP was counteracted with a low FN, and vice-versa

e. The average FN and FP are 9% and 30%, respectively

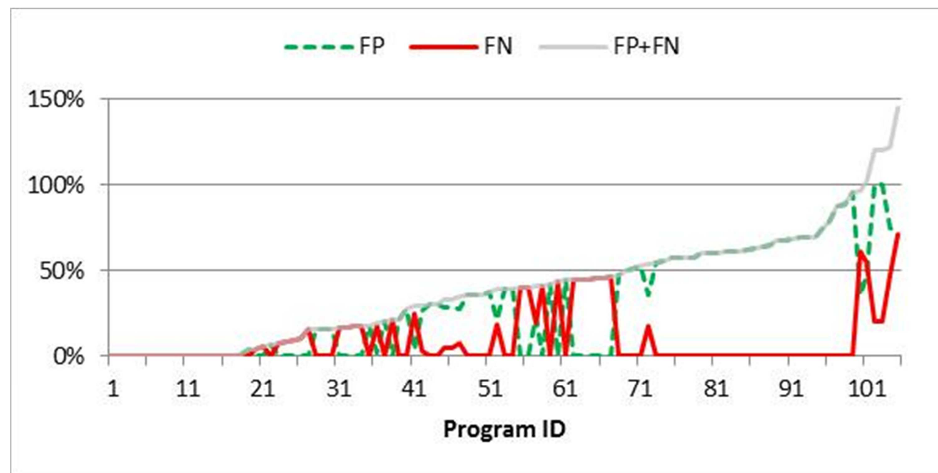



**Figure 6. CC cleansing results using *Tech-II* (BB)**

In summary, *Tech-I* is the better performer in terms of its low rate of false negatives, whereas *Tech-II* is better in terms of its lower rate of false positives. But when taking both FN and FP into consideration, the average of FN+FP is 47% for *Tech-I* and 39% for *Tech-II*, making *Tech-II* the better overall performer.

Figures 7 and 8 present the results for the 73 java versions when *ALL* profiling is applied. The same general conclusions can be made in this scenario. Specifically, *Tech-I* tends to outperform *Tech-II* in terms of false negatives while the contrary holds in terms of false positives and the aggregate of false alarms. *Tech-I* results in 3% FN and 39% FP on

average with 18 versions having a value of FN+FP equal to 0. On the other hand, *Tech-II* results in 12% FN and 19% FP on average with 11 versions (15%) having a value of FN+FP equal to 0. It's worth mentioning that the results for individual versions are almost the same when compared to the *BB* profiling scenario. Concerning *Tech-I*, the results remained exactly the same for 53 versions, were slightly different for 12 others, and varied with no particular pattern in the remaining 8 versions (i.e. didn't result in lower/higher FP/FN all the time). For *Tech-II*, the results remained the same for 46 versions, were slightly different for 19 others, and also varied with no particular pattern for the remaining ones.

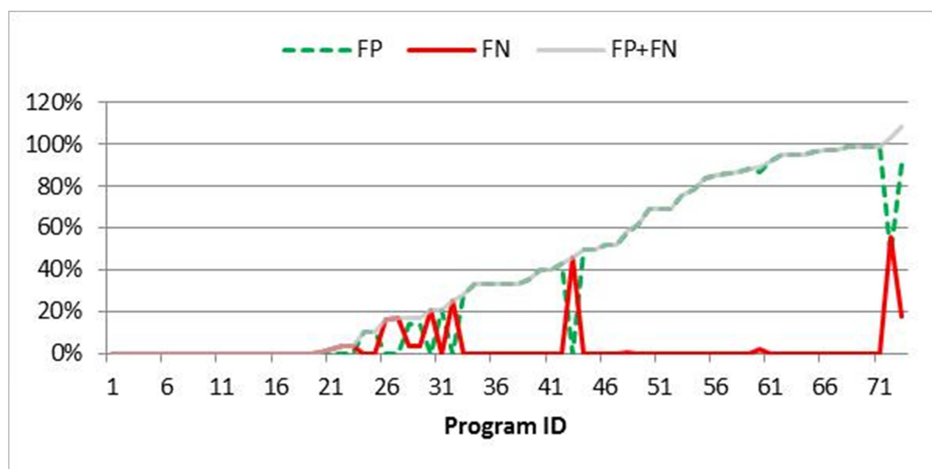



**Figure 7. CC cleansing results using *Tech-I* (ALL)**

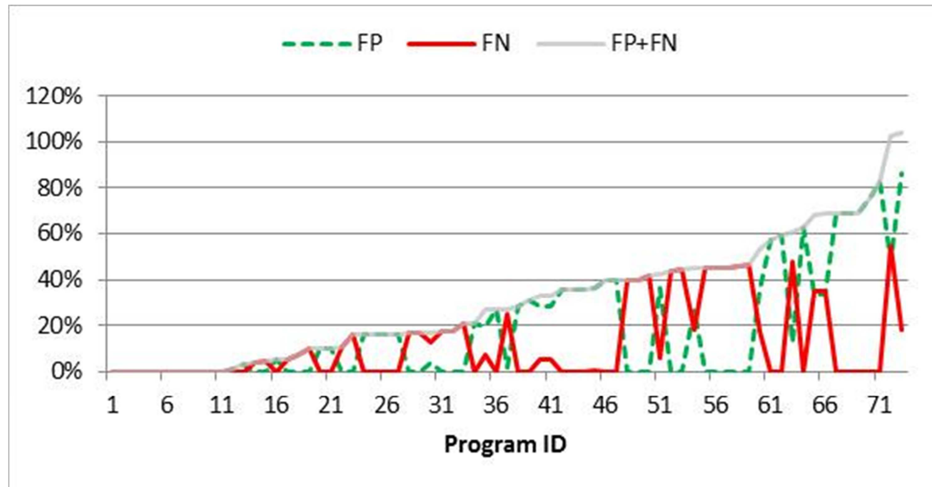



**Figure 8. CC cleansing results using *Tech-II* (ALL)**

Furthermore, we intended to apply our techniques for extreme cases in which no weak *CC* or no true passing runs are present in the test suite being considered. We used 31 versions with no *CC*'s and 6 others with no true passing runs. In the former case, *Tech-I* resulted in an average of false positives equal to 70% whereas *Tech-II* averaged 60%. Obviously, no false negatives would be expected as there are no *CC*'s to begin with. Concerning the versions with no true passing tests, the average rate of false negatives resulting from *Tech-I* and *Tech-II* are 21% and 24% respectively. Similarly, one wouldn't expect any false positives in such versions.

### *4. Analysis*

Columns 6, 7, and 8 in Table 6 show the value of *FaultScore* (using *BB* profiling) in each of the following scenarios, respectively: 1) before applying any of the cleansing

techniques, 2) after applying *Tech-I*, and 3) after applying *Tech-II*. The average improvement in *FaultScore* was 0.52 when applying *Tech-I* and 0.39 when applying *Tech-II*. Applying *Tech-I* led to improving *FaultScore* values for all 105 versions. Among these, the value of *FaultScore* reached a (maximum) value of 1 for 88 versions. Similarly, applying *Tech-II* led to improving *FaultScore* for all 105 versions and achieving a value of 1 for 67 of them.

The results of *ALL* profiling were very similar. The values of *FaultScore* after applying *Tech-I* (resp. *Tech-II*) were the same as those presented in column 7 (resp. column 8) except for 8 applications (resp. 10 applications). The average improvement was 0.53 for *Tech-I* and 0.38 for *Tech-II*. Again, the value of *FaultScore* improved for all 105 versions using both techniques. Furthermore, a *FaultScore* of 1 was obtained for 87 versions using *Tech-I* and 66 versions using *Tech-II*. These results suggest that our techniques are likely to benefit CBFL. Note that the reason that *Tech-I* appears superior in this context is probably due to the fact that it yields a lower rate of false negatives.

Concerning the suitability of *MaxScore* as an indicator of the presence of CC's, we first considered the 31 versions exhibiting no CC's, and computed their associated *MaxScore* using *T;* they turned out to be all equal to 1. Also, considering the 111 versions that did contain CC's, only 18 had a *MaxScore* of 1. This suggests that a *MaxScore* of 1 is somewhat a good predictor that *T* does not contain CC's, in which case cleansing is not needed.

Our final conclusion concerns the impact of the profiling type on cleansing efficiency. In relation to the discussions of Figures 7 and 8, among the 73 versions for

which both *BB* and *ALL* profiling was used, *Tech-I* gave almost the same results for the vast majority of them (89% of the versions) and didn't follow a particular trend for the remaining ones. These observations apply to *Tech-II* as well and therefore it can be concluded that the profiling type does not matter. Another observation that supports this claim is that *MaxScore* (before cleansing) exhibited the same values using *BB* and *ALL* profiling for all 73 Java versions.

### D. Related Work

A coverage refinement approach is presented in [125] to reduce the influence of coincidental correctness on fault localization. The work introduces a concept called *context-pattern*, which is unique for each fault type and describes the program behavior before and after the faulty code. Coverage results for all statements are refined with the context-pattern following a context-pattern matching. The paper lists context patterns for specific types of fault such as: missing function calls and missing assignments. It also presents results of experiments that showed that coincidental correctness is a common problem and harmful for coverage-based fault localization.

In the work presented in [120], which is based on [110], the authors described coincidental correctness as a problem that occurs whenever the weak mutation hypothesis (*WMH*) is not holding. *WMH* states that whenever a fault was executed and its effect is detectable at the fault location then the output will be affected. The work was primarily an empirical study to identify how often *WMH* holds. On the other hand, the work in [20] focused on identifying points in the tested code that are likely to develop the type of

undetected errors that lead to coincidental correctness. The *PIE* analysis presented in the paper investigates three factors related to faulty elements in the tested program: fault execution, creation of faulty states (infection), and failure propagation to the output. Fault execution analysis can be done by studying the probability of execution of certain locations in the code based on a predefined set of inputs. Infection can be analyzed based on a pre-estimation for a fault behavior. Finally, the propagation of fault can be studied by injecting failure states and then studying their propagation estimate.

# CHAPTER IV

# ONLINE INTRUSION DETECTION USING PROFILE-BASED SIGNATURES

Intrusions or attacks, particularly in the context of an application, make use of a *vulnerability* within the application in order to induce unsafe behavior; the input used to take advantage of this vulnerability is referred to as an *exploit*. Consequently, signatures corresponding to such attacks can be partitioned into exploit-based and vulnerability-based signatures; the former are derived from the inputs inducing the intrusion, whereas the latter are constructed using the vulnerability itself. Relying solely on exploit-based signatures in an IDS allows it to become viable to polymorphic attacks - attacks that differ syntactically but semantically trigger the same vulnerability [40]. As such, we opt to devise an intrusion detection approach that leverages signatures that are in part vulnerability-based, specifically, these signatures do not characterize the vulnerabilities themselves, but instead they characterize program events that correlate with (and not necessarily cause) the exploitation of program vulnerabilities.

We rely on program execution profiling information to define attack signatures, which we term *profile-based signatures*. In this work, execution profiles comprise method calls, method call pairs, basic blocks, branches and definition-use pairs. This approach is inspired by the idea that exploits induced by a certain vulnerability are likely to exhibit similar traits or patterns of execution, so by mining out the collection of events gathered during the occurrence of attacks, we would be able to characterize the events or program

conditions that correlate with vulnerabilities, and hence devise corresponding signatures. As a result, these signatures can be deployed within an intrusion detection system to capture future exploits, by checking if any incoming executions induce the events they describe. Note that since the identified profile-based signatures correlate with a given exploit, there exists a potential of using them to understand, locate, and fix the associated vulnerability [12][19], but this is not the concern of this work. Instead, our concern is to prevent a system from being attacked in cases when one is aware that attacks due to a certain vulnerability occurred, and the task of understanding, locating, and fixing that vulnerability is underway. It is not uncommon that such task might take a very long time or might be even abandoned altogether due to the risk of modifying the code.

It is not often the case that an attack is induced by the execution of a single program element (e.g., the execution of a single statement or a single branch), but by the execution of a combination of such elements. Consider, for example, the following code fragment:

```
s1.     x = 2;
s2.     y = -2;

s3.     if (…)
s4.             x += 10;

s5.     if (…)
s6.             y = y * -2;

s7.     y = y + -2*x;
s8.     z = x * y;
```

```
s9.     if (z >= 0) // should be if (z > 0)
s10.            access granted
s11.    else
s12.            access denied
```

The intrusion occurs only when `z` is equal to `0` at `s9`, which is the site of the *"vulnerability"* in this case, allowing access when it should be denied. This action can only take place if the definition-use pairs *DUP(s1,s7), DUP(s6,s7)* and branches *Branch(s3,s5)*, *Branch(s5,s6)* execute in a program run. Any single element of these on its own is not enough to induce the erroneous behavior, but a certain combination of them can allow for the exploitation of the vulnerability, and thereby an intrusion. For instance, detecting the execution of the branch *Branch(s3,s5)* and the def-use pair *DUP(s6,s7)*, or alternatively the def-use pair *DUP(s1,s7)* and the branch *Branch(s5,s6)*, should alert to an attack. Note that any one of these combinations is a sufficient and necessary condition for the occurrence of the intrusion, and thus any of them can be utilized as a signature in the detection system.

The main phases of our proposed solution are elaborated on in the subsequent sections. In summary, the following tasks are applied on the vulnerable subject application:

*Task1. Creating a training set that adequately characterizes the usage pattern of the application and includes both normal tests and attack tests.*

*Task2. Executing the training set and collecting the execution profiles.*

*Task3. Generating the profile-based signatures that mostly correlate with the attacks.*

Since a brute-force enumeration of all possible combinations is not a feasible

solution, this task uses a genetic algorithm to identify the combinations of program elements that correlate with the attacks that occurred within the training set.

*Task4. Incorporating the generated signature(s) into the intrusion detection system.*

The remainder of the chapter is organized as follows. We begin explaining the phases of our implementation in Section A with execution profiling, followed by signature generation in Section B, and signature matching in Section C. The empirical evaluation and analysis of obtained results is given in Section D. Finally, in Section E we comparatively discuss related work.

## A. Execution Profiling

Each test in our training set, whether it represents an attack or a normal behavior, will have a profile generated for it containing information about the occurrence of the following program elements:

1) Method calls (*MC*): For every method *M* that is executed in at least one test, an *MC* profile entry indicates whether *M* is called in the current test
2) Method call pairs (*MCP*): For every combination of methods *M1* and *M2* such that *M1* calls *M2* in at least one test, an *MCP* profile entry indicates whether *M1* calls *M2* in the current test
3) Basic blocks (*BB*): For every basic block *B* such that *B* is executed in at least one test, a *BB* profile entry indicates whether *B* is executed in the current test

4) Basic-block edges or branches (*BBE*): For every pair of basic blocks *B1* and *B2* such that there is a branch from *B1* to *B2* in at least one test, a *BBE* profile entry indicates whether this branch is taken in the current test
5) Def-use pairs (*DUP*): For every pair consisting of a variable definition $D(x)$ and a use $U(x)$ such that $D(x)$ dynamically reaches $U(x)$ in at least one test, a *DUP* profile entry indicates whether $D(x)$ dynamically reaches $U(x)$ in the current test

In most cases, a combination of these elements should be enough to characterize the behavior of the profiled application and should allow some form of distinction between safe and malicious program runs. It is this distinction that we aim to leverage. Finally, it should be noted that the profiling tool that we use in this work targets the Java platform and has been developed in prior work [60][63][65][67].

## B. Signature Generation

The goal of our work is to generate signatures that are representative of attack patterns, and comprised of combinations of program elements of multiple types. Obviously, such combinations must be of relatively minimal-size, to allow for tolerable runtime overhead (during the matching process), and have executed in as many exploits and as few safe runs as possible, so as to ensure acceptable rates of false positives and false negatives. Generating these signatures using the brute-force algorithm would entail considering all possible combinations of profiled elements, i.e. an exponential number of combinations with respect to the size of the profiles, which is not a viable solution. Therefore, an approximation algorithm is the rational alternative, and in our case, we use a genetic

algorithm. Genetic algorithms are global search heuristics used to solve combinatorial optimization and learning problems [55]. In general, a genetic algorithm solves a given problem by operating on a population of candidate solutions, evaluating their quality, then applying a form of transformation over *generations* or iterations to improve the quality of these solutions, and ultimately evolving to a single solution - or set of solutions - that fits certain criteria.

In what follows we discuss the design of the genetic algorithm as used for the purpose of generating attack-inducing signatures. Note that we generate one signature per vulnerability, even though one signature characterizing multiple vulnerabilities could also be generated using our algorithm.

*Chromosome Representation.* In our implementation, a chromosome has to represent a combination of multi-type elements, so a bit string notation would be suitable to indicate which profiled elements are included in each combination instance. The size of each bit string is equal to the total number of execution elements gathered during the profiling phase; a bit set to 1 implies that the corresponding element is included in this particular combination. Therefore, by varying the positions of 1s and 0s in a bit string, a new combination of program elements is created, and the number of 1s in the bit string corresponds to the size of the combination. An example of chromosome representation in the context of our problem is given in Figure 9. The circled 1-bit indicates that the corresponding method call element is included in the combination; the 0-bit implies that the corresponding DU-pair element is not included.

**Figure 9. Chromosome Representation**

*Fitness Function*. Once an individual chromosome is created, its fitness is evaluated to determine how good this combination solution is in identifying exploits. For this purpose, we employ a fitness function that relies on the number of safe and unsafe test cases in the training set which contain this combination. In particular, the fitness function is a numeric measure of the quality of the solution that indicates the execution frequency of a combination in malicious test cases relative to safe test cases as is shown in this equation:

$$fitness(chromosome) = \%unsafe\ runs - \%safe\ runs$$

where *%unsafe runs* is the number of malicious test cases containing the combination over the total number of malicious test cases, and likewise for *%safe runs*. Ultimately, the aim is to end up with a solution whose fitness is equal to 1, i.e. a combination (or set of combinations) that occurred in all exploits but not in any safe run. This would guarantee the elimination of false negatives and false positives from the training set, but not necessarily from all yet unseen tests, of course.

*Population Generation.* The population is a set of chromosomes signifying a collection of candidate solutions, which will evolve into the final solution. We begin with the *initiating-chromosome* from which the entire population is spawned. The initiating-chromosome is constructed as a function of the intersection of the execution profiles of all

unsafe test runs in the training set. The resulting program elements are the ones of interest since they occurred in all attacks associated with the given vulnerability. The population is then formed, one combination bit string at a time, by taking a random subset of the initiating-chromosome. The subset selection is done in a probabilistically-randomized manner: for each element of the initiating-chromosome, the chosen probability determines if it is to be included in the combination or not. The value of this probability governs the size of the resulting combination relative to the size of the initiating-chromosome, so we tend to use rather small probabilities to satisfy our requirement for minimal-sized combinations. To further illustrate the population generation mechanism, Figure 10 presents a sample of three individual chromosomes derived from the initiating-chromosome. The initiating-chromosome representing the intersection of exploit profiles is shown in 10-(a) whereas the resulting population chromosomes are shown in 10-(b); the 0-bit circled in red indicates that the element did not execute in all unsafe test case, placing a 0-bit in all individuals. The 1-bits in the initiating-chromosome are probabilistically assigned as 1s or 0s in the population chromosomes.

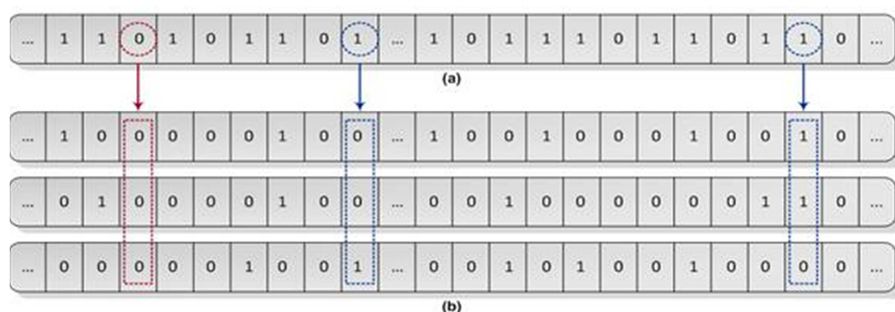



**Figure 10. Initial Population Generation**

*Transformation Operator.* We employ *fitness-based crossover* [77], its basic functioning resembles that of genetic heredity, where a new chromosome is produced as a result of combining two parent chromosomes and passing down properties from each onto the new child, always favoring the parent with the higher fitness. The intended result is for the constructed child to have as good or better fitness than its parents. The adopted genetic algorithm is a *steady-state* one, implying that the transformation is applied across generations, in each generation creating a single new child which replaces another individual in the population. To diversify the child generation process, a child chromosome is either the result of a crossover or a completely random setting similar to that introduced in the population generation phase; however, crossover is applied at a much higher rate to maintain the property of inheritance. To conduct the crossover, two parent combinations are randomly selected from the population then the child generated is a combination containing program elements from both parents, behaving similarly to the idea of inheriting traits from one's parents. Nevertheless, to ensure improvement of the child's fitness, more elements should be taken from the parent combination with higher fitness. On the bit string level, each bit in the child chromosome is set to be equal to the same-indexed-bit in one of its parents, always favoring the better-fitted one. This is accomplished by assigning the probability factor of adopting the bit value from the parent with better fitness to be higher than that of the lower-fitness parent.

*Acceptance Criterion.* The fitness function is a measure of the quality of a certain solution or combination, but to actually determine when a solution is deemed fit enough to be considered, a certain threshold for the fitness value must be set. In our case, we evaluate

the fitness of a chromosome to determine whether it is fit enough to include in the general population once it is created. However, we don't require such threshold to be high. In fact, small values are desirable as they ensure the formation of a population to start with as well as achieving more diversity among the candidate solutions.

*Stopping Criterion.* The last step to determine when the generational evolution should stop, indicating that an adequate solution to the problem has been attained. This happens when one of the following conditions is met: 1) a solution with fitness equals to 1.0 is encountered, which means that there is no room for improvement anymore; or 2) a maximum number of generations is reached. It is worth mentioning that we keep track of the best encountered solution (resulting from a crossover or random generation) throughout the entire process. This guarantees that we end up having the best alternative in case no solution with fitness of 1.0 was found.

### C. Signature Matching

The back-end of our proposed IDS involves *execution profiling* followed by *signature generation*; *signature matching* constitutes its front-end. First, it should be pointed out that signature generation is carried out for each individual vulnerability, i.e., given an application with multiple vulnerabilities, multiple independent signatures are generated, and their resulting XML representations are merged into a single definition file to be passed to the signature matching subsystem. Hereafter, a signature generated for an individual vulnerability will be referred to simply as *signature*, and the collection of signatures for all vulnerabilities in the application will be referred to as *signature-set.*

Given the signature-set definition file, the signature matching subsystem parses it at startup and maintains a description of the program elements characterizing the signatures in order to: 1) dynamically instrument the target application, 2) perform signature matching, and 3) alert in case of intrusions. This subsystem checks if all the program elements (*MC*, *MCP*, *BB*, *BBE*, *DUP*) constituting a given signature in the signature-set occur in a single execution of the application. To enable this type of surveillance, instrumentation is required. Instrumentation is the injection of instructions at the Java bytecode-level of an application. In this instance, instrumentation is meant to detect when the elements identified as part of a signature are in fact encountered. Therefore, two modules are designed to construct the signature matching subsystem, and will be discussed next: 1) the instrumentation module, and 2) the matching module.

### *1. Instrumentation Module*

In order to achieve acceptable online performance we choose to apply selective and dynamic instrumentation to enable the required application monitoring. Selective instrumentation is desirable as only the elements specified by the signatures (and some related ones) need to be analyzed [69]. The advantage of dynamic instrumentation is that it does not require stopping and restarting the deployed application any time a new signature is to be registered within the matching system, which is not the case with static instrumentation.

In implementing our module we use the java.lang.instrument package, which enables *dynamic* instrumentation and can be used in conjunction with the bytecode

manipulation library *BCEL* [156], which provides functionality to inject bytecode instructions. This package allows instrumentation by way of Java *agents*, which are pluggable libraries that run embedded in the Java Virtual Machine (JVM) and intercept the class-loading process. The agents are run in tandem with the target application and are programmed to carry out the instrumentation. The implemented agent (1) reads in the XML formatted signature-set definition file, (2) parses it to determine what particular elements are involved in each of the signatures, then (3) instruments the classes (at load time) to insert the necessary bytecode instructions. The inserted instructions which consist of calls to methods within the matching module vary according to the type of elements in a given signature; therefore, calls are inserted at the following locations:

1) For *MC* elements: at the entry statement of the method specified by the signature
2) For *MCP* elements: at the entry and exit of all methods in the application as this is needed in order to track the application's dynamic call stack. Given a signature specified as *MCP*(*f1*, *f2*), it is not sufficient to simply check whether functions *f1* and *f2* were entered, but the matching module should check the top of the call stack at the entry of *f2* to make sure that *f1* was actually the caller
3) For *BB* elements: at the entry statement of the basic block specified by the signature
4) For *BBE* elements: at the entry statement of all *BB*s in the method containing the *BBE* specified by the signature. A *BBE* entails a source *BB* and a target *BB* both of which belonging to the same method. Instrumenting the source and target of the *BBE* is not sufficient as it is not always the case that the target will execute right after the source.

5) For *DUP* elements: at the definition and use statements specified by the signature, and also at all other statements that define the involved variable.

As it is apparent from the above, the efficiency of online signature matching is dictated not only by the number of elements constituting the signatures in the signature-set, but also by the type of these elements. Clearly, the cost of monitoring *MC*s and *BB*s is lower than of *MCP*s, *BBE*s and *DUP*s. For this reason, it is beneficial to first try to identify signatures containing only *MC*s and *BB*s, and in case none are found, then try to identify signatures containing the other types of elements. In fact, this scheme is adopted in our work, as described in Section D.2.

### 2. Matching Module

The matching module maintains a flag for each of the signatures in the signature-set, as well as a set of flags for each of the elements within the given signature. If all the elements constituting a particular signature have executed and had their flags set, then the flag pertaining to that signature is set as well, and a message alerts to the detection of an intrusion. Consider the following two scenarios where the element under inspection is a *DUP(s1, s2)* involving a local variable. In the first: a) the definition site *s1* is executed leading the matching module to set a flag indicating the execution of the definition, and b) the use site *s2* is executed while the definition flag is still set. In the second: a) the definition site *s1* is executed leading the module to set a flag indicating the execution of the definition, b) the variable was redefined leading the matching module to reset the definition flag, and c) the use site *s2* is executed while the definition flag is not set as a result of b). In

the first scenario, the module will indicate that *DUP(s1, s2)* has matched, whereas in the second it will not.

## D. Empirical Evaluation

### *1. Subject Programs and Test Suites*

Our study involves seven applications, three having a security focus, namely, *Tomcat 3.0*, *Tomcat 3.2.1*, and *Jigsaw 2.0.5*, and four that do not, i.e., *print_tokens2*, *JTidy,* *schedule*, and *tot_info*. The latter applications are used to demonstrate the applicability of the approach to online failure detection; noting that most existing work on online failure detection [39][43] is based on anomaly detection. For simplicity, we will refer to the defects and failures in *print_tokens2*, *JTidy*, *schedule*, and *tot_info* as vulnerabilities and attacks, respectively.

*Apache Tomcat 3.0* is an open-source Servlet/JSP container having four vulnerabilities: JSP source disclosure (*vul-1*), directory listing disclosure (*vul-2*), and JSP engine denial of service (*vul-3* and *vul-4*). The test suite consists of 658 requests, 460 of which are safe, while the rest are attack-inducing. Note that in this program, as well as in the others, the number of malicious requests is far less than that of safe ones within the test suite, which is analogous to a real-life situation. The safe requests included 193 Servlet requests, 150 JSP requests, and 117 HTML/text requests. Whereas the unsafe requests included 150 requests exploiting *vul-1*, 38 requests exploiting *vul-2*, 6 requests for *vul-3*, and 4 requests for *vul-4*.

*Apache Tomcat 3.2.1* contains three vulnerabilities: *vul-1* exhibits JSP source code disclosure, and *vul-2* and *vul-3* exhibit JSP engine denial of service. The test suite consists of 497 requests, with only 24 unsafe tests. The safe requests include 283 Servlet requests, 69 JSP requests, and 121 HTML/text requests. The unsafe requests comprise 18 requests exploiting *vul-1,* 2 requests exploiting *vul-2*, and 4 requests for *vul-3*.

*Jigsaw 2.0.5* is also an open-source web server and Servlet container containing four vulnerabilities: denial of service (*vul-1*), path disclosure (*vul-2*), directory listing disclosure (*vul-3*) and illegal file access (*vul-4*). The test suite contains 490 normal requests and 40 exploits. The safe requests include 193 Servlet requests, 191 JSP requests, and 106 normal HTML/text requests. The unsafe requests include 1 request exploiting *vul-1,* 2 requests exploiting *vul-2,* 4 requests for *vul-3,* and 33 requests for *vul-4.*

*JTidy 3.0* is an HTML syntax checker and pretty printer. Its test suite comprises 1000 files (each containing 280 lines on average). Some of the tests were downloaded from the Google Groups (groups.google.com) using a web crawler and the others were part of the XML Conformance Test Suite. Of these, 192 were XML files and the rest were HTML files. *JTidy* failed on 180 of these test cases distributed as follows: 1) 83 exercised *vul-1*; 2) 2 exercised *vul-2*; 3) 95 exercised *vul-3*.

*print_tokens2* is a lexical analyzer developed as part of the Siemens benchmark [89]. We constructed a multiple-fault version using some of the original bugs. The test suite contains 1801 passing tests and 548 failing ones distributed among 7 vulnerabilities as follows: 1) 205 exercised *vul-1*; 2) 146 exercised *vul-2*; 3) 19 exercised *vul-3*; 4) 20 exercised *vul-4*; 5) 96 exercised *vul-5*; 6) 33 exercised *vul-6*; and 7) 29 exercised *vul-7*.

*schedule* is a priority scheduler also from the Siemens suite. The multiple-fault version we used involved 981 passing tests and 1314 failing tests. Among the failing runs, 1070 exercised *vul-1*, 30 exercised *vul-2*, and 214 exercised *vul-3*.

*tot_info* is another Siemens program, which computes information measures. We set up a multiple-fault version of it that includes 6 vulnerabilities. The test suite consists of 791 passing runs and 148 failing runs: 1) 20 exercised *vul-1*; 2) 19 exercised *vul-2*; 3) 37 exercised *vul-3*; 4) 1 exercised *vul-4*; 5) 3 exercised *vul-5*; and 6) 68 exercised *vul-6*.

### 2. *Experimental Setup*

We evaluate the effectiveness of the system in detecting attacks by measuring the rate of false positives and false negatives exhibited. We also consider the overhead imposed during its online deployment. For this purpose, we conduct experiments on our subject programs and compute the relevant metrics. Each subject program has an associated test suite *T* that, for the sake of our study, we consider to be an adequate representation of its input space. For each program we apply the following steps:

1) Identify the safe and unsafe test cases in *T*
2) Generate the execution profiles for the tests in *T*
3) For each vulnerability perform the following
   a. Construct a training set *T'* that is a subset of *T*, which includes both safe and unsafe runs. We opt to vary the size of *T'* in order to assess its effect on the accuracy of our approach, especially when only a fraction of the possible exploits is presented during the learning process. Specifically, we choose |*T'*|/|*T*| to take on the values

5%, 10%, 20%, through 90%, respectively. Also, in our experiments we explore two modes for constructing *T'*, *clustering* and *random*, described later in this section.

b. Apply the genetic algorithm to generate a signature associated with the vulnerability comprising combinations of program elements. And as mentioned previously, we will first try to generate signatures containing only *MC*s and *BB*s, and in case none are found, then we will try to generate signatures containing the other types of elements. Specifically, signature generation will be orderly conducted using the following types of elements until a high fitness signature is found: {*MC*s, *BB*s}, {*MC*s, *BB*s, *DUP*s}, {*MC*s, *BB*s, *DUP*s, *MCP*s}, {*MC*s, *BB*s, *DUP*s, *MCP*s, *BBE*s}

4) Save the resulting signature-set in an XML definition file
5) Activate the signature matching subsystem using the signature-set and rerun the application using *T*
6) Determine the number of false positives and false negatives detected
7) Measure the slowdown as the time ratio of running the test suite with and without activating the matching system

The *random* mode for constructing the training set is conducted by simply randomly selecting tests from *T* to form *T'*, whereas the *clustering* mode proceeds as follows: test cases in *T* are automatically clustered based on the similarity of their execution profiles comprising *MC*s, *MCP*s, *BB*s, *BBE*s, and *DUP*s. Then *T'* is built by randomly selecting, if possible, two tests from each cluster, one safe test and one unsafe test associated with the given vulnerability. This process is repeated over all clusters until the desired percentage of

safe/unsafe tests is reached. The goal here is for *T'* to cover, as much as possible, the behaviors induced by *T*. A similar approach was used for the purpose of test suite minimization in [67]. Note that the size of *T'* is dictated by the chosen number of clusters and by our decision to ensure that at least two unsafe test case from the given vulnerability is included.

Obviously, these two modes are not applicable in practice since *T* is unknown. We investigate them strictly to validate our assumption that the training set must adequately characterize the usage pattern of the application, as we hypothesize that *T'* built using *clustering* should characterize *T* better then when built using *random*. And thus, we expect our results to turn out better using *clustering*.

Finally, due to the non-determinism introduced by the selection of the training sets and the use of the genetic algorithm, the results presented in the following sub-section are reported by averaging the results from five iterations of our approach.

### *3. Results*

We measure the number of false alarms produced when a certain signature-set is deployed. False alarms include false negatives induced by undetected exploits and false positives resulting from incorrect labeling of safe executions. By monitoring the output of our IDS, we can determine which test cases are flagged as attacks when the IDS issues an alert. Then, this collection of *"attacks"* is compared with the actual set of exploits given initially: a missing request is tagged as a *false negative*, and any safe request in the

*“attacks”* collection is counted as a *false positive*. The percentage of false alarms is computed as follows:

$$\% \, False \, Negatives = (\# \, of \, falsely \, flagged \, exploits)/(Total \, \# \, of \, exploits)$$

$$\% \, False \, Positives = (\# \, of \, falsely \, flagged \, safe \, runs)/(Total \, \# \, of \, safe \, runs)$$

The amount of overhead induced is measured by running *T* on the application prior to instrumentation and then again after the IDS is activated, and in both cases recording the execution times. The slowdown is thus calculated as:

$$\% \, Slowdown = (Time \, with \, IDS - Time \, without \, IDS)/(Time \, without \, IDS)$$

In addition to the above three entities, we compute the following:

1) *Average Signature Length (ASL)*: average length (number of program elements) of the generated individual signatures in the signature-set
2) *Elements Types (Types)*: types of the elements constituting the signatures in the signature-set (involved in all five iterations)
3) *Best Possible Fitness using Training Set* ($BPF_{T'}$): fitness of the intersecting elements amongst the failing runs in *T'*. If $BPF_{T'} < 1.0$, our approach will *not* yield good results. This metric could be computed prior to deployment; therefore, a user will be aware of the potential risk of incoming false alarms in case its value is less than 1.0.
4) *Best Possible Fitness using Full Test Suite* ($BPF_T$): fitness of the intersecting elements amongst the failing runs in *T*. If $BPF_T < 1.0$, our approach will *not* yield good results. Note that this number cannot be computed in practice as *T* is not known; we are computing it strictly for analysis purposes.

Table 7 consists of selected entries corresponding to the cases when our approach performed best in terms of both low rates of false alarms and small sizes of the training sets. The 7 rows denoted as *All* correspond to when all vulnerabilities in a given application are considered (the whole signature-set), the other 30 correspond to when individual vulnerabilities are considered. The table presents the number of exploits/failures, then for both modes of building the training set, it shows the best observed result. We make the following observations regarding the *clustering* mode:

1) Our approach performed very well except for *vul-3* in *jigsaw*, *vul-2* in *schedule*, and *vul-2* and *vul-3* in *tot_info*. That is, for 26 out 30 cases, it exhibited no false alarms given small training sets. Note that out of the 26 successful cases, 19 involved training sets of size 5%, 3 of size 10%, 2 of size 20%, and 2 of size 30%. Also, the average signature length varied from 1.0 to 4.2 with an average of 2.14; and except for 6 cases, the signatures involved *MC*s and/or *BB*s only. This implies that the genetic algorithm was capable, in most cases, to identify small, simple and effective signatures from relatively small training sets.
2) The values of $BPF_T$ and $BPF_{T'}$ is 1 for all entries except *vul-2* in *schedule*, and *vul-2* and *vul-3* in *tot_info*, which explains why our approach performed poorly for these cases. This shortcoming is due to the fact that given the used profile types there were no combinations of elements that highly correlate with the exploits at hand. Also, given that the values of $BPF_{T'}$ are known to be less than 1.0 prior to deployment; a user of our approach could take precautionary measures to deal with the potential incoming false alarms.

3) As expected, when our approach does not perform well on one or more individual vulnerabilities it also performs poorly on the combined vulnerabilities, as exhibited in the *All* entries.
4) In regard to the slowdown induced by our matching system, it was not insignificant but not hindering as it varied from 46% to 102%. Specifically, the slowdown was 74.2%, 65.5%, 92.7%, 47%, 102%, 46%, and 60%, for *Tomcat 3.0*, *Tomcat 3.2.1*, *Jigsaw 2.0.5*, *print_tokens2*, *JTidy*, *schedule*, and *tot_info*, respectively. Note that these numbers correspond to when the whole signature-set is considered (i.e., *All*).
5) Considering *vul-3* in *jigsaw*, it is the rate of false negatives that is excessively high and not the rate of false positives. This means that when forming the training sets, the selected unsafe tests are not adequately characterizing the malicious usage pattern of the application. Also, as shown in Table 7, *random* performed unexpectedly better than *clustering*. Therefore, the poor performance here is due to an inadequate training set. Note that only 4 unsafe tests are present in *T* which is likely the reason behind this behavior.
6) In Section D.2 we expected that *clustering* would yield better results than *random*. Table 7 supports that in general except for the case of *vul-3* in *jigsaw*.

To summarize, our approach worked very well for 26 out of the 30 vulnerabilities but failed for the remaining 4. That is, in 86.67% of the cases it produced no false alarms using training sets of size 30% or less. Note that the average rates of false negatives and false positives were 0.43% and 1.03%, respectively.

**Table 7. Selected data from all applications**

| | | # Fail | Clustering | | | | | Random | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | *\|T'\|/\|T\|* | *%FP* | *%FN* | *ASL* | *Types* | *\|T'\|/\|T\|* | *%FP* | *%FN* | *ASL* | *Types* |
| **Tomcat 3.0** | *vul-1* | 150 | 5% | 0% | 0% | 2.0 | *BB* | 5% | 0% | 0% | 1.0 | *BB* |
| | *vul-2* | 38 | 5% | 0% | 0% | 1.0 | *BB* | 10% | 0% | 0% | 1.0 | *BB* |
| | *vul-3* | 6 | 30% | 0% | 0% | 3.6 | *MC-BB* | 30% | 0% | 0% | 4.0 | *MC-BB* |
| | *vul-4* | 4 | 10% | 0% | 0% | 3.6 | *MC-BB* | 5% | 0% | 0% | 3.6 | *MC-BB* |
| | *All* | 198 | 30% | 0% | 0% | - | - | 30% | 0% | 0% | - | - |
| **Tomcat** | *vul-1* | 18 | 5% | 0% | 0% | 2.8 | *MC-BB* | 5% | 0% | 0% | 1.0 | *BB* |
| | *vul-2* | 2 | 5% | 0% | 0% | 2.6 | *MC-BB* | 30% | 0% | 0% | 3.0 | *MC-BB* |
| | *vul-3* | 4 | 5% | 0% | 0% | 3.8 | *MC-BB* | 20% | 0% | 0% | 3.6 | *MC-BB* |
| | *All* | 24 | 5% | 0% | 0% | - | - | 30% | 0% | 0% | - | - |
| **Jigsaw** | *vul-1* | 1 | 5% | 0% | 0% | 3.2 | *MC-BB* | 5% | 0% | 0% | 4.0 | *MC-BB* |
| | *vul-2* | 2 | 5% | 0% | 0% | 3.2 | *MC-BB* | 5% | 0% | 0% | 3.2 | *MC-BB* |
| | *vul-3* | 4 | 10% | 0% | 10% | 3.4 | *MC-BB* | 5% | 0.08% | 0% | 1.8 | *MC-BB* |
| | *vul-4* | 33 | 5% | 0% | 0% | 2.0 | *BB* | 5% | 0% | 0% | 2.4 | *MC-BB* |
| | *All* | 40 | 60% | 0% | 0% | - | - | 5% | 0.08% | 0% | - | - |
| **JTidy** | *vul-1* | 83 | 30% | 0% | 0% | 2.0 | *BB* | 40% | 0% | 0% | 3.8 | *MC-BB* |
| | *vul-2* | 2 | 5% | 0% | 0% | 2.6 | *MC-BB* | 5% | 0% | 0% | 2.4 | *BB* |
| | *vul-3* | 95 | 10% | 0% | 0% | 3.4 | *MC-BB* | 10% | 0.02% | 0% | 3.0 | *MC-BB* |
| | *All* | 180 | 30% | 0% | 0% | - | - | 40% | 0% | 0% | - | - |
| **print_tokens2** | *vul-1* | 205 | 5% | 0% | 0% | 1.0 | *BB* | 5% | 0% | 0% | 1.0 | *BB* |
| | *vul-2* | 146 | 5% | 0% | 0% | 1.0 | *BBE* | 5% | 0% | 0% | 1.0 | *BBE* |
| | *vul-3* | 19 | 5% | 0% | 0% | 2.0 | *MC-BB* | 10% | 0% | 0% | 2.0 | *MC-BB* |
| | *vul-4* | 20 | 20% | 0% | 0% | 2.0 | *MC-BB* | 60% | 0% | 0% | 2.0 | *MC-BB* |
| | *vul-5* | 96 | 5% | 0% | 0% | 1.0 | *BB* | 30% | 0% | 0% | 1.0 | *BB* |
| | *vul-6* | 33 | 20% | 0% | 0% | 2.0 | *MC-BB* | 20% | 0% | 0% | 2.0 | *MC-BB* |
| | *vul-7* | 29 | 10% | 0% | 0% | 1.0 | *MC* | 20% | 0% | 0% | 1.0 | *MC* |
| | *All* | 548 | 5% | 0% | 0% | - | - | 5% | 0% | 0% | - | - |
| **Schedule** | *vul-1* | 1070 | 5% | 0% | 0% | 1.0 | *BB* | 20% | 0% | 0% | 1.0 | *BB* |
| | *vul-2* | 30 | 40% | 5.5% | 4.7% | 4.2 | *MC-BB-DUP* | 100% | 8.1% | 0% | 5.2 | *MC-BB-DUP* |
| | *vul-3* | 214 | 5% | 0% | 0% | 2.0 | *BB* | 10% | 0% | 0% | 2.6 | *BB* |
| | *All* | 1314 | 40% | 5.5% | 0.11% | - | - | 80% | 7.2% | 0.05% | - | - |
| **tot_info** | *vul-1* | 20 | 5% | 0% | 0% | 1.0 | *DUP* | 30% | 0% | 0% | 1.0 | *DUP* |
| | *vul-2* | 19 | 50% | 14.4% | 0% | 1.6 | *BB* | 50% | 14.4% | 0% | 2.0 | *BB* |
| | *vul-3* | 37 | 20% | 2.78% | 0% | 2.2 | *BB-BBE* | 60% | 2.78% | 0% | 3.0 | *BB-BBE* |
| | *vul-4* | 1 | 5% | 0% | 0% | 1.0 | *BB* | 5% | 0% | 0% | 2.2 | *MC-BB* |
| | *vul-5* | 3 | 5% | 0% | 0% | 1.0 | *DUP* | 10% | 0% | 6.67% | 1.2 | *MC-DUP* |
| | *vul-6* | 68 | 5% | 0% | 0% | 1.0 | *DUP* | 10% | 0% | 0% | 1.8 | *MC-DUP* |
| | *All* | 148 | 20% | 10.2% | 1.2% | - | - | 60% | 14.1% | 0% | - | - |

## *4. Profiling and Cost Analysis*

Table 8 shows the number of profiling elements recorded for each of the seven applications, categorized by element type. For example, as a result of executing the *Tomcat 3.0* test suite, a total of 26,137 distinct program elements were tracked and recorded of which 1,083 were *MC*s, 1,640 were *MCP*s, 7,130 were *BB*s, 7,533 were *BBE*s, and 8,751 were *DUP*s. Typically, these numbers differ based on the type and structure of the application, and are indicative of the size of the bit strings operated on in the genetic algorithm.

**Table 8. Number of program elements per application**

| | *Tomcat 3.0* | *Tomcat 3.2.1* | *Jigsaw* | *print_tokens2* | *JTidy* | *schedule* | *tot_info* |
|---|---|---|---|---|---|---|---|
| ***Total*** | 26137 | 24438 | 29895 | 891 | 22110 | 1047 | 1276 |
| ***MCs*** | 1083 | 1030 | 1216 | 22 | 325 | 24 | 18 |
| ***MCPs*** | 1640 | 1637 | 2155 | 30 | 693 | 37 | 23 |
| ***BBs*** | 7130 | 6485 | 7553 | 278 | 4853 | 238 | 271 |
| ***BBEs*** | 7533 | 6777 | 7978 | 310 | 5604 | 280 | 315 |
| ***DUPs*** | 8751 | 8509 | 10993 | 251 | 10635 | 468 | 649 |

*Cost of Task1*: since the training set is formed by simply augmenting the existing application's test suite by the observed attacks, the cost of creating it was insignificant in our study.

*Cost of Task2*: the cost of collecting the test suites profiles for *Tomcat 3.0*, *Tomcat 3.2.1*, *Jigsaw 2.0.5*, *print_tokens2*, *JTidy*, *schedule*, and *tot_info* was 13 minutes, 1.1 minutes, 3.6 minutes, 83 minutes, 271 minutes, 57 minutes, and 30 minutes, respectively. Noting that profiling the safe runs of a given test suite is required to be performed only once (for the

lifetime of the version of the application), whereas the attack runs must be profiled right after they get discovered in the field.

*Cost of Task3*: the signature generation process took, on average, less than one second for 26 vulnerabilities. But took considerably more for *vul-2* in *schedule* (63 sec), *vul-2* in *tot_info* (30 sec), *vul-3* in *tot_info* (20 sec), and *vul-2* in *print_tokens2* (4 sec).

*Cost of Task4*: as stated in the previous section incorporating the generated signatures in the IDS induced a showdown that varied from 46% to 102%.

## E. Related Work

In [58] Martin *et al.* presented PQL (Program Query Language), a language that allows developers to specify code patterns that characterize given vulnerabilities. The user-provided code patterns are then used to generate a static matcher and a dynamic matcher. Aspect-oriented programming is used to instrument the application in order to generate execution traces to be analyzed by the dynamic matcher. This approach resembles our proposed solution as it also monitors executing events, but differs in the following:

1. It requires the user to examine the application code and manually specify the code patterns (i.e., signatures) to be matched, which is demanding on the part of the user and prone to error. In our solution, on the other hand, the signatures are automatically discovered
2. The use of aspect-oriented programming limits the expressiveness and complexity of the user-defined patterns [71]. The patterns discovered by our technique are complex as they encompass combinations of profiling elements

In [27] Lorenzoli *et al.* presented a technique that identifies failure contexts and prevents future occurrences of the failures they describe. An outline of the technique follows:

1. The user manually specifies oracles, in the form of JML assertions, which address specific fault types
2. Static data-flow analysis is used to automatically identify the program points that potentially affect the oracles
3. The program points are monitored during training to augment the oracles with dynamically generated properties (using Daikon [5])
4. The user specified oracles and automatically generated properties are then used for failure detection and failure analysis

The methodology of our approach is fundamentally different than what Lorenzoli *et al.* proposed, and more importantly our approach has two main advantages: a) it does not require the user to define oracles; and b) it is oblivious to fault type.

Exploit-based pattern matching techniques analyze incoming input to extract a pattern to be matched against attack sequences stored in the detection system's database. The pattern could be a sequence of bytes, or a combination of entities, e.g., Kim and Karp [51] used patterns involving the IP protocol number, the destination port number, and the sequence of bytes. This methodology falls short at detecting variants of a given exploit, which made some researchers shift their focus towards vulnerability-based pattern matching [40][41]. Our proposed pattern matching technique is neither exploit-based nor vulnerability-based, but is more similar to the latter than to the former since our generated signatures characterize the program behavior induced by the vulnerability.

Brumley *et al.* [40] introduced the concept of a vulnerability signature which is a representation for the set of inputs that satisfy a specified vulnerability condition. Given a new detected exploit for an unknown vulnerability, and the tuple ($P$, $T$, $x$, $c$) where $P$ is the program, $x$ is the exploit string, $c$ is a vulnerability condition satisfied by $x$ (e.g., "heap overflow at a specific line number"), and $T$ is the execution trace of $P$ on $x$. The aim is to generate a vulnerability signature that will match future malicious inputs $x'$ by examining them without running $P$. Our technique is more advantageous as it does not require any information about the vulnerability, whereas their technique hinges on detailed information, namely, the vulnerability condition and location.

In [70] Newsome and Song proposed the use of dynamic taint analysis for the automatic generation of exploit-based signatures. They implemented *TaintCheck*, a tool that enables the user to mark (taint) untrusted inputs to be tracked via dynamic data-flow analysis, in order to detect whether it is used to carry out an attack. *TaintCheck* allows the detection of attacks that cause sensitive program values to be overwritten with the attacker's data, i.e., *overwrite attacks*. Following the detection of an attack, the tool provides information that could be used to automatically generate signatures to be deployed for attack filtering. Our proposed technique has the following advantages: 1) it is more general as it is not limited to overwrite attacks; 2) it does not require the user to pinpoint the inputs to be tainted; and 3) it is noted in [70] that *TaintCheck* slowed down the target application between 1.5 to 40 times, the overhead imposed by our implementation is much lower as it varied from 46% to 102%.

# CHAPTER V

# USER-DEFINED COVERAGE CRITERION FOR TEST CASE INTENT VERIFICATION

In practice, program correctness is mainly affirmed through *testing*, i.e., by checking that the program produces the expected output. *Regression testing* is an essential part of the maintenance phase of software development; its goal is to ensure that the behavior of existing code, believed correct by previous testing, is not altered by new program changes. Since exhaustive testing is not feasible, testers rely on *coverage criteria* to guide their test selection and to provide a stopping rule for testing.

We argue that the primary focus of regression testing should be on code associated with: a) earlier bug fixes; and b) particular application scenarios considered to be important by the developer or tester. Existing coverage criteria do not enable such focus, e.g., 100% branch coverage does not guarantee that a given bug fix is exercised or a given application scenario is tested. Therefore, there is a need for a new and complementary coverage criterion in which the user can define a test requirement characterizing a given behavior to be covered as opposed to choosing from a pool of pre-defined and generic program elements. We propose this new methodology and call it *UCov*, a *user-defined coverage criterion* [9] wherein a test requirement [21] is an execution pattern of program elements and predicates. Our proposed criterion is not meant to replace existing criteria but to complement them as it focuses the testing on important code patterns that could go untested otherwise.

*UCov* supports test case intent verification. For example, following a bug fix, the testing team may augment the regression test suite with the test case that revealed the bug. Evidently, this new test case induces an execution pattern associated with the bug; however, it might become obsolete due to code modifications not related to the bug. But our coverage criterion, based on a user defined execution pattern (a test requirement) characterizing the bug and coupled with the test case, would:

a) Detect whether the test requirement was satisfied or not.
b) Determine whether test case intent verification passed or failed.
c) Deem the test suite deficient in case test intent verification failed. Thus, suggesting that a new test case that satisfy the requirement needs to be (manually) generated.

Current coverage criteria limit the user to choosing from a set of program elements that vary in the level of granularity and complexity. Those include statements, branches, logic expressions [82], def-uses [24], information flow pairs [63], slice pairs [60], and paths [84]. At first, it might appear that what we are proposing is simply to cover more complex test requirements comprised of some patterns or combinations of existing program elements. But in fact, the main goal and contribution of our methodology is to cover *behaviors* as opposed to generic structural program elements, and to couple tests with intents to be verified and preserved. Noting that, to our knowledge, neither of these concepts has been previously proposed, and as Sections B and D demonstrate, they fill in an important gap lacking in existing coverage criteria.

We implemented our methodology for the Java platform in a tool that provides the following:

a) *An Eclipse plugin to enable users to easily define test requirements*.

b) *The ability of cross referencing the test requirements across subsequent versions of a given program*, which is a non-trivial task due to the code differences between versions.

c) *The ability to determine whether the test requirements are satisfied*, which entails instrumenting the *System Under Test* (*SUT*) at the byte code level.

We applied *UCov* onto two real life case studies; the first case study involves a bug fix, and the second is a scenario of significance to program requirements.

The main advantages of *UCov* to the software maintenance process are described below:

a) *Bug resurrection happens when faulty code that was fixed, gets introduced again*. Typically this might happen due to the uncoordinated access of a file in a source control system by more than one developer. *UCov* ensures the coverage of the test requirement associated with the bug fix and thus uncovers the resurrecting bug. Without *UCov*, resurrecting bugs might escape typical structural-coverage-based testing.

b) *A Bug fix could become faulty due to other code changes* (i.e., a bug was introduced in the bug fix). Here also, *UCov* can detect that the test requirement associated with the bug fix is not satisfied, which calls for revisiting the bug fix and test suite.

c) In *UCov*, a test case *t* that was coupled with a bug fix, a feature, or some scenario of interest to the tester/developer, is intended to verify an expected (correct) behavior of the application. But if *t* becomes obsolete, that expected behavior would go unverified, which will be detected by *UCov*.

d) Understandably, even full coverage achieved by existing structural coverage criteria does not establish that all (or any) of the scenarios of a given algorithm are tested. To generalize item c); in *UCov*, each scenario could be coupled with a test case, thus relying on *UCov* to ensure coverage of the scenarios. This enables *validation testing* whose aim is to exercise the functionality of the *SUT*.

The remainder of this chapter is organized as follows. Section A provides definitions and notation for specifying test requirements. Section B walks through a motivating example. Section C describes the main components of *UCov*. Section D presents our real life case studies. Finally, related work is surveyed in Section E.

## A. Definitions and Notations

This section provides definitions for entities relevant to *UCov*, and notation for specifying test requirements.

*Definition 1 -* A *program element* is a basic programming unit such as a statement, a branch, or a definition-use pair.

*Definition 2 -* A *test requirement* is an execution pattern that a *test case must satisfy* or cover.

*Definition 3 -* A *basic test requirement* (*btr*) is a test requirement involving only a set of program elements and a logical expression that describes their execution. For example, basic test requirement $[(s_1 \vee b_1) \wedge (\neg dup_1)]_{btr}$, which involves the set of program elements $\{s_1, b_1, dup_1\}$, is considered to be satisfied if: a) statement $s_1$ or branch $b_1$ did execute, and,

b) definition-use pair $dup_1$ did not execute. Note that the logical operators supported by *UCov* are, negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$).

*Definition 4* - A *conditional test requirement* (*ctr*) is a test requirement comprising a test requirement *tr*, and a predicate *p* specifying a state of some program variables. For a conditional test requirement to be satisfied, *tr* should be satisfied, and *p* should evaluate to *true* immediately before. For example, the conditional test requirement $[[s_1 \wedge b_1]_{btr}, x > y]_{ctr}$ requires that statement $s_1$ and branch $b_1$ be executed and, when that happens, *x* be strictly greater than *y*.

*Definition 5* - A *sequential test requirement* (*str*) is a test requirement composed of a sequence of at least two test requirements that must be satisfied one after the other. For example, the sequential test requirement $[<[b_1]_{btr}, [b_2]_{btr}, [b_3 \vee s_1]_{btr}>]_{str}$ requires that branches $b_1$ and $b_2$ be sequentially executed, followed by $b_3$ or $s_1$.

*Definition 6* - A *repeated test requirement* (*rtr*) is a test requirement comprising a test requirement *tr*, and a range indicating the number of times it should be repeated. For example, the repeated test requirement $[[s_1 \wedge b_1]_{btr}, 5, 1000]_{rtr}$ requires that statement $s_1$ and branch $b_1$ be executed at least 5 times and at most 1000 times. In case one or both of the bounds do not matter, a "*don't care*" symbol could be specified, e.g., $[[s_1]_{btr}, 100, _\ ]_{rtr}$ requires that statement $s_1$ be executed at least 100 times.

## B. Motivation

Typically, algorithms are presented while stressing the prime scenarios they support, which we believe should all be tested for quality assurance. Noting that even full coverage achieved by existing structural coverage criteria does not establish that all (or any) of the scenarios of an algorithm are tested, we advocate our user-defined coverage criterion as an effective solution to this task. Intuitively, each documented scenario (or case) associated with the algorithm describes at least one execution pattern that should be coupled with designated test cases. We illustrate the usage of *UCov* in testing the algorithm for deleting a node in a binary search tree.

The algorithm shown in Figure 11 is presented in [87] and considers four cases concerning the node $z$ to be deleted:

*Case1* If $z$ has no children, then it is replaced by NIL.

*Case2* If $z$ has only one child, then it is replaced by that child.

*Case3* If $z$ has two children, then it is replaced by its successor, which is the leftmost node in the sub-tree rooted at the right child of $z$. In this case, the successor of $z$ (say $y$) has no right child. That is, $y$ would be a leaf and thus deleting $z$ would be achieved by replacing the contents of $z$ by those of $y$ and replacing $y$ with NIL.

*Case4* Similarly to *Case3*, $z$ has two children, and is replaced by its successor. However, here $y$ has a right child, and the contents of $z$ are replaced by those of $y$ but instead of replacing $y$ with NIL, it is replaced by its right child.

```
BST-DELETE(T, z)
Input: Binary Search Tree (T), pointer to the node to be deleted (z)
Output: Binary Search Tree (T') obtained from T by deleting z

1.  if left[z] = NIL or right[z] = NIL
2.     then y ← z
3.  else y ← TREE-SUCCESSOR(z)

4.  if left[y] ≠ NIL
5.     then x ← left[y]
6.  else x ← right[y]

7.  if x ≠ NIL
8.     then p[x] ← p[y]

9.  if p[y] = NIL
10.    then root[T] ← x
11. else if y = left[p[y]]
12.    then left[p[y]] ← x
13. else right[p[y]] ← x

14. if y ≠ z
15.    then key[z]← key[y]
16.           copy y's satellite data into z
```

**Figure 11. Pseudo-code for deleting a node in a BST**

Figure 12 depicts a test suite *T* comprising the four test cases $t_1$, $t_2$, $t_3$, and $t_4$. Table 9 details the individual and cumulative statement and branch coverage information for each of the test cases. As shown, *T* achieves 100% statement coverage and 100% branch coverage.

The execution patterns associated with each of the algorithm's scenarios are also shown at the bottom of Table 9, along with *T*'s coverage information. Test cases $t_1$ and $t_2$ cover the execution patterns (test requirements) of *Case1* and *Case2*, respectively. And both $t_3$ and $t_4$ cover the execution pattern of *Case3*. Therefore, *Case4* is left untested, i.e., none of the tests cover test requirement $[<[s_3]_{btr}, [s_6]_{btr}, [s_8]_{btr}>]_{str}$.

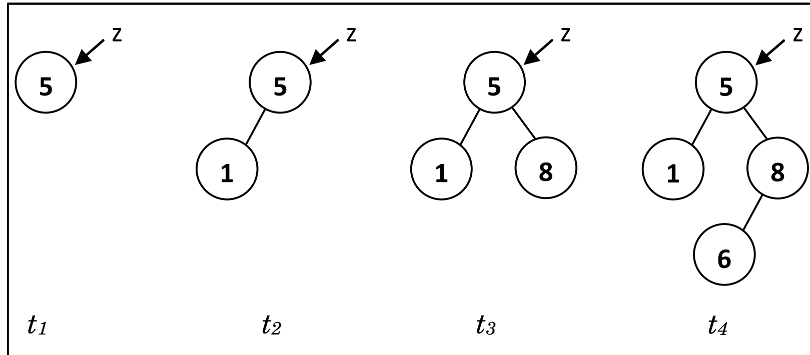



**Figure 12. Test suite T = {t1, t2, t3, t4}**

**Table 9. Coverage information for test suite T**

| | | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $\{t_1, t_2, t_3, t_4\}$ |
|---|---|---|---|---|---|---|
| **Statements** | S1 | ✓ | ✓ | ✓ | ✓ | ✓ |
| | S2 | ✓ | ✓ | × | × | ✓ |
| | S3 | × | × | ✓ | ✓ | ✓ |
| | S4 | ✓ | ✓ | ✓ | ✓ | ✓ |
| | S5 | × | ✓ | × | × | ✓ |
| | S6 | ✓ | × | ✓ | ✓ | ✓ |
| | S7 | ✓ | ✓ | ✓ | ✓ | ✓ |
| | S8 | × | ✓ | × | × | ✓ |
| | S9 | ✓ | ✓ | ✓ | ✓ | ✓ |
| | S10 | ✓ | ✓ | × | × | ✓ |
| | S11 | × | × | ✓ | ✓ | ✓ |
| | S12 | × | × | × | ✓ | ✓ |
| | S13 | × | × | ✓ | × | ✓ |
| | S14 | ✓ | ✓ | ✓ | ✓ | ✓ |
| | S15 | × | × | ✓ | ✓ | ✓ |
| | S16 | × | × | ✓ | ✓ | ✓ |
| **Branches** | S1→S2 | ✓ | ✓ | × | × | ✓ |
| | S1→S3 | × | × | ✓ | ✓ | ✓ |
| | S4→S5 | × | ✓ | × | × | ✓ |
| | S4→S6 | ✓ | × | ✓ | ✓ | ✓ |
| | S7→S8 | × | ✓ | × | × | ✓ |
| | S7→S9 | ✓ | × | ✓ | ✓ | ✓ |
| | S9→S10 | ✓ | ✓ | × | × | ✓ |
| | S9→S11 | × | × | ✓ | ✓ | ✓ |
| | S11→S12 | × | × | × | ✓ | ✓ |
| | S11→S13 | × | × | ✓ | × | ✓ |
| | S14→S15 | × | × | ✓ | ✓ | ✓ |
| | S14→END | ✓ | ✓ | × | × | ✓ |
| **Prime Scenarios** | **Execution Patterns (*TR*)** | | | | | |
| ***Case1*** | $[<[s_2]_{btr}, [s_6]_{btr}, [[s_7]_{btr}, x==NIL]_{ctr}>]_{str}$ | ✓ | × | × | × | ✓ |
| ***Case2*** | $[<[s_2]_{btr}, [s_8]_{btr}>]_{str}$ | × | ✓ | × | × | ✓ |
| ***Case3*** | $[<[s_3]_{btr}, [s_6]_{btr}, [[s_7]_{btr}, x==NIL]_{ctr}>]_{str}$ | × | × | ✓ | ✓ | ✓ |
| ***Case4*** | $[<[s_3]_{btr}, [s_6]_{btr}, [s_8]_{btr}>]_{str}$ | × | × | × | × | × |

This example demonstrates how applying our coverage criterion would deem test suite *T* deficient despite the fact that it satisfied full statement and branch coverage. In order to test all four scenarios using *UCov*, the user would: 1) specify their four respective test requirements shown at the bottom of Table 9; and 2) for each test requirement, design at least one test case that satisfies it.

## C. Methodology and Implementation

*UCov* entails three main tasks and associated components that we describe next.

### *1. Specifying Test Requirements*

We first designed and built a programming interface that enables the user to specify test requirements of the types we described in Section A. Note that our implementation expects the program elements to be specified at the Java bytecode level. Since such interface is only adequate for users who are also developers, we used a graphical Eclipse plugin that hides its complexity. The output of the plugin is compilable code that specifies the user-defined test requirements using calls to the programming interface.

Figures 13-15 present the class diagrams of the programming interface. A set *TR* of user-defined test requirements comprises a list of basic test requirements (*btr*'s), conditional test requirements (*ctr*'s), sequential test requirements (*str*'s), and repeated test requirements (*rtr*'s). Figure 13 shows that: 1) the *str*'s, and *rtr*'s are made up of any of the four types of test requirements; and 2) the *ctr*'s are made up of any of the four types of test requirements in addition to a *predicate*. Figure 14 shows that a *btr* is composed of the

conjunction, disjunction, and negation of primitive *btr*'s, which in turn are made up of statements, def-use pairs, and branches. Finally, Figure 15 shows that a *predicate* is made up of the conjunction, disjunction, and negation of primitive predicates or clauses.

To illustrate the use of *UCov*'s programming interface, the test requirement $[[s_4]_{btr}, result==true]_{ctr}$ associated with function reset shown in Figure 16 would be specified as shown in Figure 17.

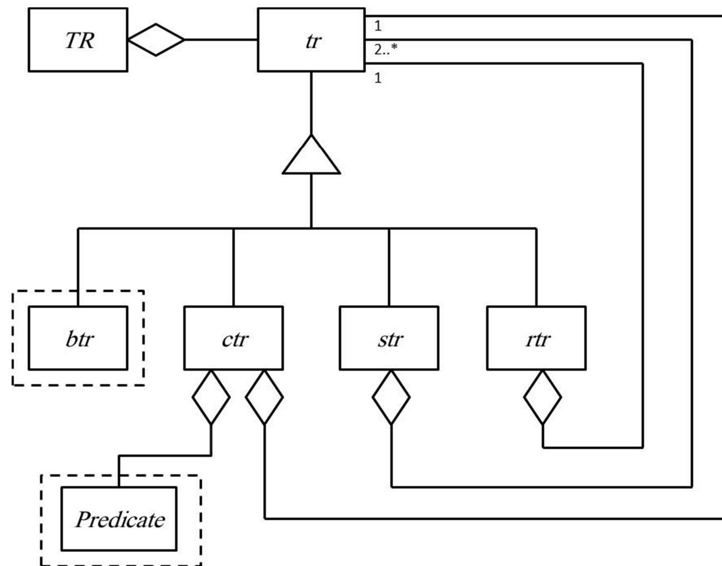



**Figure 13. Class Diagram of the programming interface**

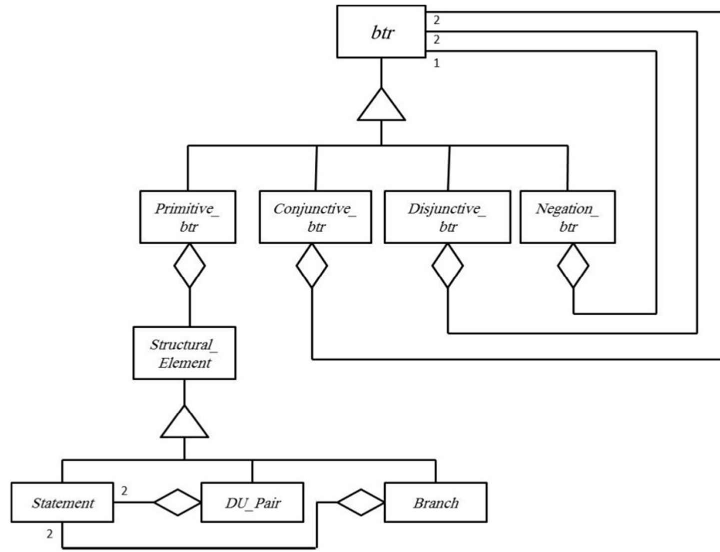



**Figure 14. Class Diagram associated with btr class**

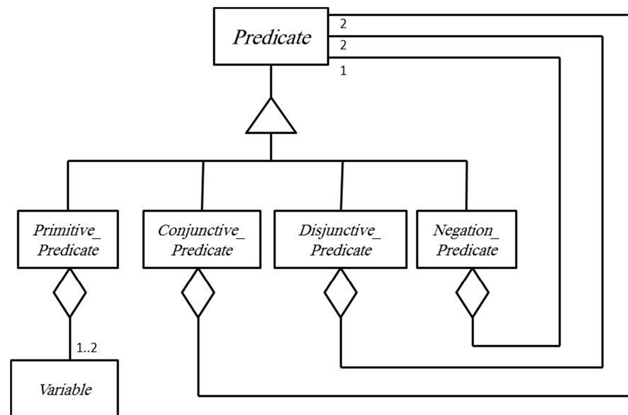



**Figure 15. Class Diagram associated with Predicate class**

```
boolean reset()

{

s1:        boolean result = false;
s2:        if (override || valveClosed)
s3:                result = true;
s4:        return result;
}
```

**Figure 16. Function to control the shutdown system in a reactor**

```
Statement s1 = new Statement("Reactor", "reset", "()Z", 19);
btr btr1 = new Primitive_btr(s1);
Variable var1 = new Variable("Local", "result", "Reactor", "reset", "()Z");
Predicate pred1 = new Primitive_Predicate("Equal",  var1, new Boolean(true));
ctr ctr1 = new ctr(btr1, pred1);
```

**Figure 17. Specifying conditional test requirement example**

## *2. Migrating Test Requirements across Versions*

The test requirements considered in *UCov* are built from statements, branches, def-uses, and predicates. Since branches and def-uses are constructed from statements, and predicates are constructed from program variables, the task of migrating test requirements across versions boils down to cross referencing statements and variables across versions.

### a. <u>Statement Mapping</u>

As our implementation targets the Java platform, we opt to match bytecode statements across versions using a technique inspired from the notion of *Abstract Syntax*

*Tress* [22][85][94][97]. Given a bytecode statement *s* defined relative to the start of a method *M* in a particular version of the software being considered, our technique identifies the counterpart of *s* relative to the start of *M* in a subsequent version by analyzing what we call the *bytecode dependence tree* (*BDT*) of *M* in both versions.

The *BDT* of a particular function is constructed statically from its list of bytecode instructions $\{s_1, s_2, \ldots, s_n\}$ as follows:

- The tree has *n+1* nodes: a root node labeled "start" and *n* descendant nodes each of which corresponds to one of the bytecode instructions.
- A node *n* is the parent of another node *n'* if one of the following holds:
    - *n* and *n'* respectively correspond to bytecode instructions $s_i$ and $s_j$ such that $s_i$ consumes a value (from the JVM's operand stack) that was produced by $s_j$. This captures the *direct data dependence* relationship described in [65].
    - *n* is either the "start" node or a node corresponding to a conditional instruction and *n'* represents a non-producer instruction in the direct scope of *n*. This captures the *direct control dependence* relationship described in [65].
- Siblings are ordered according to their relative positions in the bytecode instruction list.

Figure 18 illustrates the above by showing a snippet of Java code, the corresponding bytecode instructions, and the resulting *BDT*. The nodes of the *BDT* are annotated with the offsets of the corresponding bytecode instructions.

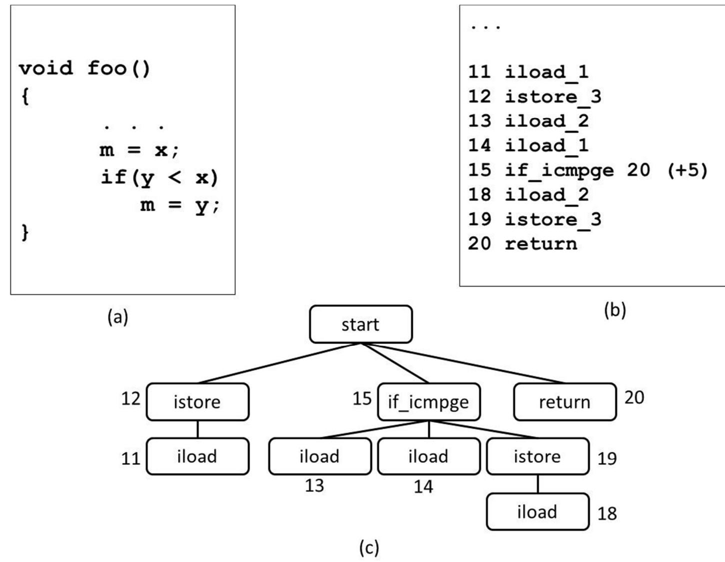



**Figure 18. (a) Sample method foo, (b) Bytecode list of foo, (c) BDT of foo**

When trying to match a test requirement *tr* against a subsequent version, every statement *s* in *tr* is mapped. We first check if the code of the method corresponding to *s* (say *M*) has changed between the two versions. If so, we construct the *BDT* of *M* with respect to the original version (say *B*) and that corresponding to the subsequent version (say *B'*). Then, we determine the node in *B'* that is structurally most similar to *s* in *B* using an iterative algorithm as follows:

1. We start with a set of potential candidates. These are the nodes in *B'* whose corresponding bytecode instruction opcode is equal to that of *s.*
2. We repeatedly eliminate the candidates which fail a similarity test of increasing precision. The order we follow is: level-1 descendants, level-1 ancestors, level-2 descendants, level-2 ancestors, level-3 descendants etc.

That is, in the first iteration, we eliminate all the candidates whose children (i.e. level-1 descendants) in *B'* are not similar to the children of *s* in *B*. In the second iteration, we eliminate (from the remaining candidates) those whose parents (i.e. level-1 ancestors) in *B'* aren't similar to the parent of *s* in *B*, and so on. And if more than one candidate still remain; we consider the siblings of *s*.

3. The algorithm successfully stops when only one highly similar candidate remains.
4. If at some iteration the set of candidates becomes empty, we restore the results of the previous iteration and require the intervention of the user to resolve the ambiguity. We also require user intervention in case we reached the final iteration with several candidates. However, both scenarios are unlikely to occur.

To demonstrate our mapping mechanism, we consider an "updated" version of the method foo of Figure 18. In the new version, shown in Figure 19 with its corresponding bytecode and *BDT*, foo is modified by adding a statement that computes the sum of *x* and *y*. In addition, variable *m* is renamed to *min* (to be revisited). We will denote the *BDT* of Figure 18 by *B* and that of Figure 19 by *B'*. Also, we will identify every node using its offset relative to the corresponding *BDT*, (for example *B-11* refers to node 11 in *B*). In what follows, we show how our algorithm maps *B-19* to *B'-29*, i.e., "m = y" in *B* to "min = y" in *B'*. We start with the set of all potential candidates; these are the nodes in *B'* associated with an *istore* instruction, the type of instruction *B-19* is associated with. Therefore, the

initial set of candidates consists of *B'-19*, *B'-22*, and *B'-29*. In the first iteration, *B'-19* is eliminated because its child differs from that of *B-19*. Alternatively, *B'-22* and *B'-29* are kept as they pass this first similarity test. In the second iteration, these two candidates undergo the second similarity test which compares their parents with that of *B-19*. As a result, *B'-22* is eliminated and *B'-29* is left as the only candidate. As such, the algorithm terminates by mapping *B-19* to *B'-29*.

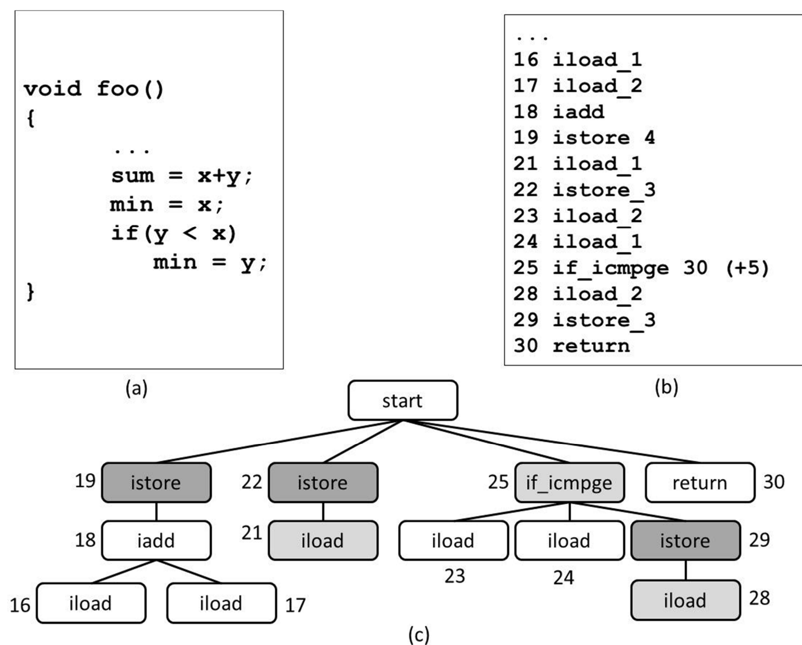



**Figure 19. Updated version of foo**

b. <u>Variable Mapping</u>

Our cross referencing technique accounts for variable matching as well. The need for this kind of matching arises when the name of a variable involved in a test requirement

is changed in the subsequent version. We leverage the statement mapping mechanism described above as a basis for variable mapping as follows:

1. For each variable *v* to be mapped, we identify the set of bytecode statements referencing it in the original version, say $S=\{s_1, s_2, \ldots, s_k\}$.
2. Then, we perform statement mapping to get the set $S'=\{s_1', s_2', \ldots, s_k'\}$ relevant to the subsequent version.
3. For each $s_i'$, we identify the variable it references and then we consider the counterpart of *v* to be the variable referenced by all statements in *S'*.
4. As described in step 3, a "perfect" match occurs when all the statements in *S'* reference the same variable. But if this was not the case, user intervention will be required in order to update the test requirement.

As an example of variable mapping, consider method foo and its updated version shown in Figures 18 and 19, respectively. The fact that variable *m* was renamed to *min* (and it is involved in a test requirement) necessitates applying the variable mapping algorithm described above. We first identify the nodes that reference *m* in *B*, which are *B-12* and *B-19*. Applying the statement mapping procedure, we map *B-12* to *B'-22* and *B-19* to *B'-29*. Then, we determine the variable(s) referenced by *B'-22* and *B'-29*. In this case, both nodes reference variable *min*, meaning that the algorithm was successful at perfectly matching *m* and *min*.

### *3. Checking the Coverage of Test Requirements*

Our approach for checking the coverage of test requirements is to some extent similar to what we adopted in [8], which was described in chapter 4, for the purpose of matching attack signatures. The approach entails two steps: *instrumentation* and *matching*, both of which are done at run-time. For a given program *P* associated with a set of user-defined test requirements *UTR*, the instrumentation module applies dynamic instrumentation at class load time on *P* to enable the online matching of the test requirements specified in *UTR*. The instrumentation is done by inserting method calls to the matching module at specific locations in *P*. These locations include:

1. Every statement specified in *UTR*. Note that, in case *UTR* was specified in a previous version, statements are mapped according to the approach discussed in Section C.2.
2. The entry statement of each method specified in *UTR.*
3. Every basic block (*BB*) leader in the method containing a branch specified in *UTR.*
4. The definition and use statements of each def-use pair (*DUP*) specified in *UTR* as well as all statements that define the variable involved.

The matching module, on the other hand, keeps track of all the *btr*'s specified in *UTR* as independent test requirements or as part of more complex ones. For every such *btr*, the matching module also maintains a *timestamp* and a *counter* indicating the last time and the number of times it got executed, respectively. In case *UTR* contains *ctr*'s, the matching module would keep track of the "current" values of all involved variables. The matching

module is triggered in two cases: 1) state update notification; and 2) structural notification. The first occurs when a variable relevant to *UTR* gets updated. In this case, the value of the corresponding variable is simply updated. The second case occurs when a *btr* referenced by *UTR* gets executed by the program. Here, the matcher updates the timestamp and the counter of the corresponding *btr* and checks all relevant test requirements.

## D. Case Studies

### *1. Testing a Bug Fix*

This case study involves two versions of *NanoXML*, an *XML* parser comprising 7,646 lines of code. The two versions were downloaded along with their test suites from the *SIR* repository [157] and they correspond to versions 1 and 3 in *SIR*. Hereafter, we will refer to these versions as *NanoXML_v1* and *NanoXML_v3*.

A typical *NanoXML* test case involves running a java test program that takes in a certain *XML* file as input and applies some *NanoXML* functionalities on it. Specifically, the test program in our case study is *Parser1_vw_v1.java* and the input file is *testvw_29.xml* shown in Figures 20 and 21, respectively. Basically, The program parses the input file using the *parse()* method defined in *StdXMLParser.java* in the *NanoXML* package and outputs the result.

This test case reveals one of the bugs in *NanoXML_v1* which is fixed in *NanoXML_v3,* namely, a **while** replaced by an **if** in method *elementAttributesProcessed* in *NonValidator.java*, shown in Figure 22. Figure 23 contrasts the faulty output against the expected output.

```
public class Parser1_vw_v1
{
    public static void main(String args[]) throws Exception
    {
        if (args.length == 0) {
            System.err.println("Usage: java Parser1_vw_v0 file.xml");
            Runtime.getRuntime().exit(1);
        }
        String filename = args[0];
        IXMLParser parser = XMLParserFactory.createDefaultXMLParser();
        IXMLReader reader = StdXMLReader.fileReader(filename);

        parser.setReader(reader);

        XMLElement xml = (XMLElement) parser.parse();
        (new XMLWriter(System.out)).write(xml);
    }
}
```

**Figure 20. Test program Parser1_vw_v1.java**

```
<!DOCTYPE FOO [
    <!ENTITY % extParamEntity SYSTEM "E:\Nanoxml\inputs\nano1\include.ent">
    <!ENTITY value "%extParamEntity;">
    <!ELEMENT FOO (#PCDATA)>
    <!ATTLIST FOO
        x CDATA #REQUIRED
        y CDATA #FIXED "fixedValue"
        z CDATA "defaultValue">
]>

<FOO x='1'>
<VAZ>vaz</VAZ>&value;</FOO>
```

**Figure 21. Input file testvw_29.xml**

```
public void elementAttributesProcessed(String     name,
                                       String nsPrefix,
                                       String nsSystemId,
                                       Properties extraAttributes)
{
  Properties props = (Properties) this.currentElements.pop(); //s1
  Enumeration _enum = props.keys();                                     //s2
  if (_enum.hasMoreElements()) //s3 -- should be while(_enum.hasMoreElements())
  {
    String key = (String) _enum.nextElement();               //s4
    extraAttributes.put(key, props.get(key));                //s5
  }
}
```

**Figure 22. Faulty code in NonValidator.java**

```
<FOO x="1"
z="defaultValue">
    <VAZ>vaz</VAZ>
    INCLUDE
</FOO>           a)
```

```
<FOO x="1" z="defaultValue"
y="fixedValue">
    <VAZ>vaz</VAZ>
    INCLUDE
</FOO>                      b)
```

**Figure 23. (a) Faulty output; and (b) Expected output**

Note that the bug fix would be exercised in *NanoXML_v3* only if the **while** loop iterates twice or more. This behavior could be captured in *UCov* by the repeated test requirement $[[s_4]_{btr}, 2, _\, ]_{rtr}$, or the sequence test requirement $[<[s_4]_{btr}, [s_4]_{btr}>]_{str}$, and possibly others.

*UCov* revealed that when executed in *NanoXML_v3*, the test case {Parser1_vw_v1.java, testvw_29.xml} did not actually exercise the bug fix (i.e., our user-defined test requirement was not covered), but instead resulted in an exception being thrown. Thus, in this real life case study, *UCov* alerted us that the test case associated with the bug fix became obsolete and that an alternate test case needs to be created.

To further investigate this case study, we manually tracked down the code change which rendered that test case obsolete and found out that it is related to the use of a different constructor of the *URL* class in method *openStream* in *StdXMLReader.java*. Noting that if the new constructor is replaced by the original one, $[[s_4]_{btr}, 2, _\, ]_{rtr}$ and $[<[s_4]_{btr}, [s_4]_{btr}>]_{str}$ would then be covered. The original code and the modified one are shown in Figure 24.

| Code | |
|---|---|
| ```<br>public Reader openStream(String publicID,<br>                         String systemID)<br>     throws MalformedURLException,<br>            FileNotFoundException,<br>            IOException<br>{<br>    systemID = "file:" + systemID;<br>    URL url = new URL(systemID);<br>    . . .<br>``` | (a) |
| ```<br>public Reader openStream(String publicID,<br>                         String systemID)<br>     throws MalformedURLException,<br>            FileNotFoundException,<br>            IOException<br>{<br>    URL url = new URL(this.currentSystemID, systemID);<br>    . . .<br>``` | (b) |

**Figure 24. Code change that renders test case {Parser1_vw_v1.java, testvw_29.xml} obsolete. (a) NanoXML_v1; and (b) NanoXML_v3**

## *2. Testing Scenarios of an Algorithm*

This case study targets the situation where a specific behavior needs to be tested. The application being considered is *tot_info*, one of the seven Siemens programs [89] that are widely used in the literature. More specifically, we inspect function *InfoTbl* that computes Kullback's information measure of a contingency table according to the following formula [92]:

$$\sum_{i=1}^{r}\sum_{j=1}^{c} x_{ij}\log(x_{ij}) - \sum_{i=1}^{r} x_i \log(x_i) - \sum_{j=1}^{c} x_j \log(x_j) + NlogN$$

where *r* and *c* are respectively the number of rows and columns in the contingency table, $x_{ij}$ is the value of the entry at row *i* and column *j*, $x_i$ is the sum of row *i*, $x_j$ is the sum of column *j*, and *N* is the sum of all entries in the table.

*InfoTbl* determines the information measure of a contingency table *T* by computing the four components of the formula above according to the pseudocode shown in Table 10. The algorithm starts by checking if *T* has at least two rows and two columns; if not, it returns -3 indicating that the table is too small. Lines 5-15 loop over the rows of *T*, compute the sum of each row and store it in array `xi`. At the same time, the sum N of all entries in the table is computed. If a negative entry is encountered during this process, the algorithm returns the "error" value -2. It also returns -1 if the total sum isn't strictly positive (lines 16-18). Similarly, the column sums are computed and stored in array `xj` (lines 19-25). The rest of the code computes each of the four components of the Kullback formula and aggregates the result in variable `info` as indicated in the table.

We distinguish three conditional checks in the code that prevent the algorithm from computing log(0). Those are the ones at lines 28, 32, and 38. The first checks if the sum of the $i^{th}$ row is different than zero, the second checks if T[i,j] is different than zero, and the third checks if the sum of the $j^{th}$ column is different than zero. We argue that an important scenario to be covered is one in which the contingency table satisfies the following four conditions:

1) Is valid, i.e., has at least 2 rows and 2 columns, doesn't have negative entries, and isn't all zeros.
2) Has at least one row whose sum is zero.
3) Has at least one column whose sum is zero.
4) Has a strictly positive information measure so that a simple contingency table such as $\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ with zero information measure would not be considered as a candidate.

We deem this scenario important because each of the three conditional checks on lines 28, 32, and 38 would evaluate both to *true* and *false* within the same test case. Applying *UCov*, the following test requirement, denoted by $\mathscr{P}$, captures the scenario at hand:

$$\mathscr{P} \equiv [<[[s_{13}]_{btr}, sum{=}{=}0]_{ctr}, [[s_{24}]_{btr}, sum{=}{=}0]_{ctr}, [[s_{42}]_{btr}, info{>}0]_{ctr}>]_{str}$$

For example, test cases based on the contingency table $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 3 \\ 0 & 1 & 1 \end{bmatrix}$ satisfy $\mathscr{P}$ and thus are deemed important.

To verify whether *tot_info*'s original (full) test suite, which was downloaded from *SIR* [157], covers $\mathscr{P}$, we created a modified version of *tot_info* in which we hard-coded some monitoring instructions that trigger a notification in case $\mathscr{P}$ is exercised. After running the modified version under the full test suite, we found out that no test case covers $\mathscr{P}$.

This real life case study shows that some scenarios that might be deemed important can go untested if not represented by non-generic test requirements such as those supported by *UCov*.

**Table 10. Information Measure Algorithm**

```
Input: Contingency table T, # rows r, # columns c
Output: -3 if r≤1 or c≤1, -2 if T contains a negative entry, -1 if T is all zeros,
Kullback measure of T otherwise

1. if r≤1 OR c≤1
2.    return -3
3. end if
4. N = 0
5. for i=1 to r
6.    sum = 0
7.    for j=1 to c
8.       if T[i,j] < 0                              Computing row sums
9.             return -2
10.      end if                                                    and
11.      sum += T[i,j]
12.   end for
13.   xi[i] = sum
14.   N += sum                                              total sum
15. end for
16. if N≤0
17.   return -1
18. end if
19. for j=1 to c
20.   sum = 0
21.   for i=1 to r
22.      sum += T[i,j]
23.   end for                                  Computing column sums
24.   xj[j] = sum
25. end for
26. info = N×log(N)        /*** 4th component of Kullback's formula ***/
27. for i=1 to r
28.   if xi[i]>0
29.      info -= xi[i]×log(xi[i])   /*** 2nd component ***/
30.   end if
31.   for j=1 to c
32.      if T[i,j]>0
33.            info += T[i,j]×log(T[i,j])/*** 1st component ***/
34.      end if
35.   end for
36. end for
37. for j=1 to c
38.   if xj[j]>0
39.      info -= xj[j]×log(xj[j])   /*** 3rd component ***/
40.   end if
41. end for
42. return info
```

## E. Related Work

Over the years, researchers have proposed numerous coverage criteria many of which are discussed or listed in [21]. The fundamentals of data flow testing and def-use coverage were presented in [93][95]. Data flow testing was contrasted against control flow and branch testing in [23] and [89]. Coverage of logical expressions is treated in [82] and [90]. Test case selection and prioritization is discussed in [153], and surveyed in [31]. However, none of the above proposed techniques is capable of verifying or preserving the intent of test cases.

Several techniques surveyed and compared in [109] aim at linking faults to test cases, and at ranking test cases based on their relevance to detected faults based on coverage metrics. These techniques employ statistical metrics and target fault localization. *UCov* differs in that it aims at establishing and *maintaining* the link between the fault, the test case, and the bug fix.

User defined coverage for hardware designs was introduced in [88] as a methodology to annotate hardware logic written in VHDL or Verilog with coverage events. The method is not intended to preserve the intent of specific test cases and is limited to hardware designs. *SystemVerilog* [86] supports a functional coverage specification language that introduces concepts like cover points, cover expressions, cover groups, and cross cover. Those coverage specifications are limited to hardware designs, are not related to specific test cases, and require knowledge of the whole design.

DSD-Crasher [98] aims at finding bugs by dynamically extracting invariants that describe the intended behavior of the program, excluding unwanted values from the domain

of the program, exploring execution paths of the program that cover the invariants, and then generating test cases that cover the extracted paths. The work does not maintain the link between the detected invariants, the extracted paths, and the test cases. *UCov* can make use of the techniques proposed in DSD-Crasher to automatically extract execution paths and link them to existing test cases after the approval of the user.

# CHAPTER VI

# LOSSLESS REDUCTION OF EXECUTION PROFILES

Execution profiles are pivotal to several dynamic program analysis techniques such as test suite minimization and prioritization [67], fault localization [12], and application-based intrusion detection [64]. The number of elements that a typical execution profile contains is usually in the order of thousands or more, amongst which a considerable proportion are redundant.

A well-known approach to reduce the high dimensionality and redundancy in execution profiles is to use *Principal Component Analysis* (*PCA*) [83][142][145]. *PCA* reduces the dimensionality of a data set (possibly involving correlated variables) to a new set involving uncorrelated variables. The generated uncorrelated variables are called *principal components* (*PC*s). The obtained set has the *PC*s ordered by the fraction of the total information/variation each retains. That is, the first PC captures as much of the variability present in the data set as possible, the second PC also captures as much of the variability but under the constraint of being uncorrelated with the previous (first) PC, and similarly for the subsequent PCs. After applying PCA, only the first few PCs are retained and the remaining ones ignored. Note that PCA transforms the original data to a *new* coordinate system. Therefore, it is not possible to recover the non-redundant profiling elements from the retained PCs, which renders many software analysis techniques

inapplicable. For example, in coverage based fault localization [6][12][19], identifying failure-correlated PCs will *not* lead to the failure-correlated program elements, which are normally needed to locate the fault. The approach we are proposing does not suffer from this limitation as it preserves the original coordinate system.

In this chapter we present an evolutionary approach for reducing redundancy in execution profiles associated with a given test suite [10]. It uses a *genetic algorithm* to find a minimal subset of profiling elements that *safely* represent the original set. The advantage of such approach isn't limited to reducing the cost of program analyses that utilize the execution profiles directly. There are situations where a limited number of variables or statements need to be identified for tracking and monitoring. One example is online intrusion and failure detection [8], another is state-based comparison of test cases [11].We evaluated our approach by applying it on eight sets of execution profiles and measuring its impact on clustering and test suite minimization. The results showed marginal deterioration in either analyses even though the reduction rate ranged from 94% to 99%.

The rest of this chapter is organized as follows. Section A presents an example that motivates our approach. Section B describes our approach. Section C describes our experimental studies and results. Finally, Section D presents a state-based comparison technique and shows how it can benefit from the proposed reduction mechanism.

## A. Motivating Example

First we show an example demonstrating how redundancy reduction is performed by identifying sets of *single* profiling elements that exhibit the same pattern of occurrence

in all the test cases. Then we demonstrate how further lossless reduction could be performed by considering subsets of the profiling elements, the subject of this work.

Table 11 shows the execution profiles for five test cases with respect to seven programming elements; e.g., statements and/or branches. For example, the first row indicates that test case $t_1$ executes the elements $e_1$, $e_3$, $e_4$, $e_6$, and $e_7$. We refer to this representation as the *Execution Matrix*.

**Table 11. Sample Execution Profiles**

| | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ | $e_7$ |
|---|---|---|---|---|---|---|---|
| $t_1$ | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| $t_2$ | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| $t_3$ | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| $t_4$ | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| $t_5$ | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

It can be noticed that $e_1$, $e_3$, and $e_7$ exhibit the same pattern of occurrence in all five test cases; similarly, $e_4$ and $e_6$. Therefore, $\{e_1, e_3, e_7\}$ could be replaced by $e_1$, and $\{e_4, e_6\}$ by $e_4$, to yield reduced execution profiles comprising only 4 elements, namely, $\{e_1, e_2, e_4, e_5\}$. We refer to this type of reduction as *Basic Redundancy Removal*.

However, this subset still contains some redundancy. Specifically, knowing the execution status of subset $\{e_1, e_4\}$ in any test case, we can infer the execution status of the remaining profiling elements. Table 12 summarizes this relationship. For example, it indicates that every test case that exercises $e_4$ but not $e_1$ exercises both of $e_2$ and $e_5$. Following this observation, the execution profiles in Table 11 could be reduced down to only two elements, $e_1$ and $e_4$.

**Table 12. {e2,e5} are inferred from {e1,e4}**

| $\{e_1,e_4\}$ | $\{e_2,e_5\}$ |
|---|---|
| 01 | 11 |
| 11 | 00 |
| 00 | 11 |
| 10 | 01 |

As demonstrated in the above example, given a set of execution profiles, our aim is to devise a method for determining a minimal subset of elements whose execution status implies the execution status of the remaining ones. In an ideal situation our algorithm should consider every subset (combination) of elements. But given their large count, such a brute force approach is infeasible, which calls for the use of an approximation algorithm, e.g., a genetic algorithm.

## B. Proposed Approach

Assuming the set of profiling elements being considered is *E*, the problem of determining the most representative subset in terms of execution status translates to the task of finding a subset S that results in the minimum conditional entropy $H(\bar{S}/S)$ where $\bar{S} = E - S$. This is equivalent to finding a subset S having a maximal value for $\frac{|DV(S)|}{|DV(E)|}$ where *DV(S)* (resp. *DV(E)*) is the set of distinct values assumed by *S* (resp. *E*). Of course, E is one such subset but we are interested in those who size is minimal. Note that the values of a given subset of elements can be regarded as the concatenation of its corresponding bits in the *Execution Matrix*. Given the size of the search space at hand ($2^{|E|}$), we opt to use a

heuristic approach to search for potential representative subsets. Specifically, we use a genetic algorithm where each candidate solution is represented by a vector of bits (*chromosome*) whose size is equal to the total number of elements. A value of 1 means that the corresponding element is included in the solution and a value of 0 indicates otherwise. The fitness of a particular solution/subset $S$ is quantified as $fitness(S) = \frac{|DV(S)|}{|DV(E)|}$. The pseudocode shown in Figure 25 describes our technique which takes an execution matrix *M* associated with a set of profiling elements *E* as input and determines a (likely small) subset of *E* with fitness equal to 1. As a pre-processing step, we perform basic redundancy removal to arrive at a reduced matrix *M'* and element set *E'*, which is useful to reduce the search space. The genetic algorithm first creates an initial population by randomly generating small subsets of *E'*. After that, it repeatedly applies crossover and mutation to produce new solutions. Every time a superior solution emerges, it replaces a less fit one in the population. This iterative process is terminated in two cases: 1) a solution having a fitness of 1 is encountered or 2) the maximum number of iterations is exhausted. In case the GA terminates without arriving at a fitness of 1, we augment the best encountered solution by adding one element at a time in a greedy fashion so as to reach the maximum fitness. One might argue that such step could be done starting from an empty solution or a totally random one. However, such approach has two disadvantages. First, it's very costly if the starting solution has low fitness. Second, as the size factor isn't enforced, this approach wouldn't likely yield small solutions. Next we detail the steps of our algorithm.

*Line 1:* Basic redundancy removal is applied on *M* and *E*. That is, *E* is partitioned into $E_1$, $E_2$, …, $E_n$ where each $E_i$ contains elements of *E* having equivalent columns in *M*.

*E'* is formed by choosing one element from each $E_i$ and *M'* is derived from *M* by removing the columns corresponding to the elements in *E-E'*.

*Lines 2-3:* Some variables are initialized; e.g., `result` is used to keep track of the best encountered solution.

*Line 4:* The initial population is built by generating random subsets of *E'* whose sizes are close to $log_2|DV(E)|$. Such choice is guided by the fact that the size of the smallest possible representative subset having a fitness of 1 cannot be smaller than this threshold. This step is an important factor to converge subsequently to a relatively small solution. The size of the initial population, which is maintained in subsequent iterations, is equal to POP_SIZE.

```
Input: Execution Matrix M, Profiling Elements E
Parameters: MAX_ITERATIONS //num of GA iterations
            POP_SIZE //population size
            MUT_PROB //probability of mutation
Output: A chromosome representing a subset of E with
fitness equal to 1

1. (M', E') ← BasicRedundancyRemoval(M, E)
2. nbIterations ← 0
3. result ← null
4. population ← genRandomSubsets(E',POP_SIZE)
5. while [nbIterations < MAX_ITERATIONS] AND
        [fit(result) < 1.0]
6.    (p1, p2) ← rouletteWheelSelection(population)
7.    child ← crossover(p1, p2)
8.    child ← mutation(child, MUT_PROB)
9.    replace(population, p1, p2, child)
10.   best ← getSolutionWithMaxFitness(population)
11.   if fit(best) > fit(result)
12.      result ← best
13.   end if
14.   nbIterations ← nbIterations + 1
15. end while
16. while [fit(result) < 1]
17.      result ← result ∪ maxGain(result, E'-result)
18. end while
19. return result
```

**Figure 25. Genetic Algorithm for Lossless Reduction**

*Line 5:* The algorithm loops MAX_ITERATIONS times unless a solution with fitness 1.0 is encountered.

*Line 6:* Within each iteration, the algorithm selects two parent chromosomes using the roulette wheel methodology, which randomly selects one chromosome at a time based on its relative fitness with respect to the population. i.e., solutions with higher fitness values are more likely to be selected.

*Line 7:* The selected parent chromosomes undergo single-point crossover to create a child chromosome as follows. First, each of them is split at the same random position. Then, a new (child) chromosome is created by concatenating the first part of the first parent with the second part of the second parent.

*Line 8:* The child chromosome undergoes mutation, where each bit is randomly flipped with a probability equal to MUT_PROB.

*Line 9:* The child chromosome replaces the less fit parent if the fitness of the former is higher than that of the latter.

*Lines 10-12:* The result is constantly updated by comparing it to the best solution obtained in every subsequent generation.

*Lines 16-18:* If the solution returned by the GA (say $S_{GA}$) doesn't have a fitness of 1.0., we augment it by adding one element at a time from $E'-S_{GA}$ until the fitness becomes 1.0. Each time we add the element that results in the maximum increase in fitness.

Concerning the parameters of the *GA*, we used a value of 1000 for MAX_ITERATIONS, 0.05 for MUT_PROB, and 100 for POP_SIZE.

## C. Experimental Study

### 1. *Methodology*

We evaluated our reduction technique by examining its impact on two common types of analysis that make use of execution profiles: clustering and test suite minimization. The main criterion was to check whether reducing the profiling elements using the proposed approach has a negative impact on the quality of the results. Cluster analysis has been used in several areas of dynamic software analysis and was applied onto execution profiles comprising program elements that varied from statements to slice pairs [60] and paths [144]. In this work we will adopt a straightforward clustering approach that uses *k-means* with Euclidean distance measures. Test cases would be represented as binary vectors where the $i^{th}$ bit is set to 1 (resp. 0) if the corresponding profiling element is exercised by it (resp. not exercised). On the other hand, test suite minimization techniques aim at finding a minimal subset that covers the same elements as the original suite. A typical approach is to select one test at a time in a greedy manner so as to maximize the number of covered elements. We will refer to this approach by *GTSM* (Greedy Test Suite Minimization).

### 2. *Subject Programs and Profiling Types*

Our experiments involved 5 Java applications (*tot_info*, *schedule, schedule2, replace,* and *print_tokens2*) and 3 programs written in C (*space*, *flex2*, and *sed3*). We downloaded all 8 applications from the SIR repository [157] and seeded each of them with 2 to 5 faults (also included in the SIR packages). After augmenting each application with oracles and executing the associated test suite, we were able to label each test case as

passing or failing. Failing test cases were further labeled according to the fault triggered. As a result, for each application we had $N_F$+1 *class labels* where $N_F$ denotes the number of faults. Note that we discarded all failing tests that triggered more than one bug.

We used the tool developed in [67] to instrument the Java programs where the generated execution profiles consisted of basic blocks, branches, method calls, method call pairs, and def-use pairs. Concerning the *C* programs, we generated execution profiles consisting solely of statements using the *GCOV* tool [155].

### 3. Metrics

For each subject program, we applied *k-means* clustering and *GTSM* before and after reduction. The quality of clustering was quantified using Rand Index [146] and F-measure [32], both of which evaluate the similarly between the clustering obtained by a given algorithm (*k-means* in our case) to the "perfect" clustering where elements belonging to the same (resp. different) class are placed in the same (resp. different) cluster. Concerning *GTSM*, we assessed its quality in either scenario by counting the number of class labels retained in the minimized suite. As such, the value of this metric ranges between 1 and the total number of class labels, with higher values indicating better performance.

### 4. Results

Table 13 shows the results we obtained for the 8 subject programs. Columns 2, 3, and 4 respectively correspond to the number of profiling elements before reduction, those after reduction, and the resulting reduction rate. Column 5 compares the F-measure for the

*k-means* clustering when applied on the original element set as opposed to the reduced one. Similarly, column 6 contrasts the Rand index for both cases. The last column enumerates the total number of class labels, those retained by applying *GTSM* on the original profiles, and those retained by applying it on the reduced ones. Note that all the experiments involving non-determinism (i.e. genetic algorithm and/or *k-means*) were repeated 10 times and the results were averaged accordingly (values in columns 3 and 7 were rounded to the nearest integer). The main conclusion that can be drawn is that our reduction mechanism has no significant negative impact on either analysis despite the fact that the reduction rate ranged between 94% and 99%. More specifically, the F-measure and the Rand index were nearly the same for all programs except flex2 where the difference in the F-measure was higher than the average. A possible explanation might be the extent of reduction that reached 99%. Concerning *GTSM*, for three of the programs (*tot_info, schedule2,* and *space*), the number of labels captured by the minimized suite didn't change after reduction. For the other four (*schedule, replace, print_tokens2,* and *sed3*), fault coverage was reduced by 1 after reduction. As for flex2, the coverage was reduced by 2. Again, the last result might also be attributed to the high reduction rate.

It's worth mentioning that the original (unreduced) data turned out to be unsuitable for clustering and minimization in the first place. This is evident from the generally low values of F-measure and Rand index as well as the fact that in 6 out of the 8 programs *GTSM* missed at least one fault. However, our goal wasn't to discuss the quality of clustering or *GTSM* but rather to show that applying our reduction technique doesn't deteriorate such analyses.

**Table 13. Results**

| Program | # Elements | # Elements Post -Reduction | % Reduction | F-measure orig/GA | Rand orig/GA | # Labels All/orig/GA |
|---|---|---|---|---|---|---|
| *tot_info* | *1269* | *45* | *96%* | *0.23/0.22* | *0.46/0.45* | *6/4/4* |
| *schedule* | *1043* | *63* | *94%* | *0.09/0.08* | *0.28/0.28* | *4/3/2* |
| *schedule2* | *1288* | *61* | *95%* | *0.07/0.07* | *0.06/0.06* | *4/3/3* |
| *Replace* | *901* | *39* | *96%* | *0.19/0.2* | *0.08/0.08* | *3/2/1* |
| *print_tokens2* | *879* | *48* | *95%* | *0.25/0.24* | *0.3/0.3* | *5/4/3* |
| *Space* | *3164* | *142* | *96%* | *0.07/0.07* | *0.1/0.1* | *6/5/5* |
| *flex2* | *2914* | *31* | *99%* | *0.2/0.15* | *0.46/0.45* | *4/4/2* |
| *sed3* | *1328* | *37* | *97%* | *0.3/0.29* | *0.35/0.35* | *4/4/3* |

## D. Benefits for State-based Comparison of Test Cases

We presented in [11] three metrics to quantify the dissimilarity between state-based execution profiles. In what follows we present the proposed approach and explain how it's likely to benefit from the lossless reduction mechanism investigated in this chapter.

State profiling requires the ability to capture the values taken by the program variables at some points during execution. For this purpose, using the *Byte Code Engineering Library* (BCEL) [156], we built a tool capable of capturing the values assigned to variables in a Java program. By default, this is done at every definition of each program variable. But the tool is made configurable to enable the user to select which variables to monitor and when to capture their values. Limiting the number of variables and the sampling rate is crucial for practicality reasons, given that realistic programs contain a large number of variables (unbounded in some cases) and are possibly long running.

At first hand, monitoring the inputs and outputs seems intuitive, but it might not be as beneficial as monitoring some internal variables that are more likely to characterize the

defect. Therefore, in situations where the user of our tool is familiar with the code (possibly the developer), internal variables deemed critical to the implementation should be selected for monitoring. However, when such option is not available, we can benefit from the lossless reduction mechanism presented previously as follows: 1) use reduction to identify a minimal subset of structural elements, 2) identify the variables associated with such elements, and 3) monitor the values of the variables identified in step 2).

Once the variables to be monitored are selected, the tool could capture their values following every definition (assignment). But the user can choose other sampling times, e.g., at the exit of specific functions, or at the execution of specific statements.

### *1. Dissimilarity Metrics*

Execution profiles are typically compared in pairs to build a dissimilarity matrix that contains the dissimilarity metrics values between all profiles. In this work we adopt and evaluate the dissimilarity metrics described below. Note that ($x_i$) and ($y_i$) correspond to the sequences of values assumed by the same monitored variable in both profiles

*Euclidean:*

$$\sqrt{\sum (x_i - y_i)^2}$$

*Chi-Square Goodness of Fit:*

$$\sum \frac{(x_i - y_i)^2}{x_i}$$

*Relative Entropy:*

$$\sum (x_i * \log\left(\frac{x_i}{y_i}\right))$$

For all the metrics, when comparing two profiles of unequal lengths we prolonged the size of the shorter profile by duplicating its last value. Also, in order to compare the shapes of the functions represented by the profiles as opposed to comparing exact values, we normalized each profile to range between 0 and 1.

***2. Results***

In order to evaluate the proposed dissimilarity metrics, we conducted experiments using 10 versions of 4 programs from the Siemens benchmark (*print_tokens*, *print_tokens2*, *schedule*, and *tot_info*). For each seeded version and each metric type we computed the metrics values between all pairs of distinct tests that satisfy the following: 1) both tests are passing, denoted as category *PP*; and 2) one test is passing and the other is failing, denoted as category *PF*. We then computed and compared the average metrics values for each of the

2 categories. It should be noted that in our experiments the profiles comprised the values assigned at every definition of a single variable that we considered critical to the implementation. The values are captured within a specific function, possibly at more than one location. We found that in 8 out of 10 cases, using all three metrics, passing tests were on average most dissimilar from failing tests. We also observed that the *Euclidean* and the *Chi-Square* metrics agreed on all cases. These results clearly demonstrate that state-based comparison is promising.

# CHAPTER VII

# CONCLUSIONS AND FUTURE WORK

## A. Thesis Contributions

We investigated several approaches to overcome the limitations of certain software analysis techniques that rely on analyzing execution profiles. We proposed modeling the runtime behavior of software via complex structures that involve state profiling, dependence information, combinations of structural elements, and sequence data. Our work involved various areas such as fault localization, regression testing, and intrusion detection.

In chapter 2 we proposed a fault localization technique based on identifying short dependence chains that are highly correlated with failure. In addition to considering data and control dependences, we augment each chain by computing a set of predicates involving the source values and target values of its edges. We used 18 versions of the Siemens test suite to evaluate the effectiveness of our technique in comparison to when statement coverage is used. Our results were promising as the technique successfully identified more correlated chains in 17 out of 18 versions.

In chapter 3 we showed that coincidental correctness is prevalent and demonstrated that it is a safety reducing factor for coverage-based fault localization. We then proposed two techniques for cleansing test suites from coincidental correctness to enhance the safety of CBFL, given that the test cases have already been classified as failing

or passing. We evaluated the effectiveness of our techniques by empirically quantifying their accuracy in identifying CC tests. The results were promising, e.g., the better performing technique, using 105 test suites and basic-block coverage, exhibited 9% false negatives and 30% false positives on average, and no false negatives nor false positives in 14.3% of the test suites. Using 73 test suites and ALL coverage (combined basic-block, basic-block-edges, and def-use pairs), the numbers were 12%, 19%, and 15%, respectively. This work also allowed us to conclude that our techniques are likely to benefit coverage-based fault localization since applying them always led to a higher suspiciousness score for the fault.

In chapter 4 we proposed an approach to application-based intrusion detection relying on profile-based signatures. Our technique starts by collecting exercised program elements, including method calls, method call pairs, basic blocks, basic block edges, and definition-use pairs. The actual construction of the needed signature is a learning process, implemented via a genetic algorithm, which generates from a training set of executions the combinations of elements that lead to unsafe executions. These combinations form the signature, which is fed to a matching system that monitors the application by way of dynamic selective instrumentation and alerts to intrusions when a match is detected. Our experiments involved seven Java applications containing 30 vulnerabilities/defects. The results showed that our approach worked very well for 26 vulnerabilities/defects (86.67%) and the overhead imposed by the system is somewhat acceptable as it varied from 46% to 102%. The exhibited average rates of false negatives and false positives were 0.43% and 1.03%, respectively.

Chapter 5 presented *UCov*, a methodology and tool support for precise test case intent verification in regression test suites. *UCov* complements existing coverage criteria by focusing the testing on important code patterns or behaviors that could go untested otherwise. That is, *UCov* allows the tester to specify user-defined test requirements to be covered. Such requirements incorporate structural elements, predicates describing the state of select program variables, and sequence information.

Chapter 6 presented a lossless reduction mechanism intended to mitigate the impact of high dimensionality present in most types of execution profiles. We evaluated this approach by applying it on eight sets of execution profiles, and our results showed a reduction rate from 94% to 99% without significant deterioration in the quality of clustering and test suite minimization. We also showed how state-based comparison of test cases could benefit from such reduction mechanism.

## B. Future Work

We plan to refine *UCov* by extending the set of predicates that could be used for specifying test requirements. The current version allows for first-order logic predicates only, which is not enough to describe all the states that might be of interest to the user. Also, we intend to investigate test case intent preservation. That is, in case of a failed test intent verification, automated test case generation will be performed whose aim is to satisfy the user-defined test requirement and thus preserve the intent of the test case.

Concerning intrusion detection, we will explore using test case generation tools to automatically create new inputs. This might further improve the performance of the *IDS*

because the latter is dependent on the quality of the training sets. We believe that 3 out of the 4 cases in which our approach performed poorly, are preventable and can be addressed in future work. In these cases, signatures that characterize the exploits could not be found because we only relied on structural program elements to build our execution profiles. To remedy this shortcoming, we plan to incorporate state information in the process of signature generation by augmenting statements or other structural elements with simple predicates. This type of augmented profiles could also be investigated in the context of identifying coincidentally correct test cases.

We also intend to leverage state profiling for automatic classification of test cases and test suite minimization. We envision a supervised learning mechanism that utilizes a neural network to carry out the classification task. The values of select internal variables at specific events, such as the execution of certain statements, would be used as input to the neural network. As for test suite minimization, we will explore a greedy approach that determines a minimal set of test cases whose “state” coverage is equivalent to the original test suite.

# REFERENCES

[1] T. Xie and D. Notkin, "Checking inside the black box: Regression testing by comparing value spectra," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 869 – 883, 2005.

[2] T. Xie, D. Marinov, and D. Notkin, "Rostra: A framework for detecting redundant object-oriented unit tests," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pp. 196 – 205, 2004.

[3] P. A. Francis, Advanced Techniques for Software Failure Classification and Observation-based Testing. PhD thesis, Case Western Reserve University, 2005.

[4] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 199 – 209, 2011.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99 – 123, 2001.

[6] R. Abou Assi and W. Masri, "Identifying failure-correlated dependence chains," in *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 607 – 616, 2011.

[7] W. Masri and R. Abou Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 1, pp. 8:1 – 8:28, 2014.

[8] 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.

[9] R. Abou Assi, F. Zaraket, and W. Masri, "Ucov: a user-defined coverage criterion for test case intent verification," *Submitted to ACM Transactions on Software Engineering and Methodology*, 2014.

[10] R. Abou Assi and W. Masri, "Lossless reduction of execution profiles using a genetic algorithm," in *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2014.

[11] W. Masri, J. Daou, and R. Abou Assi, "State profiling of internal variables," in *Proceedings of the 7th IEEE International Conference on Software Testing, Verication and Validation Workshops*, 2014.

[12] 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*, pp. 467 – 477, 2002.

[13] T. Denmat, M. Ducasse, and O. Ridoux, "Data mining and crosschecking of execution traces," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 396 – 399, 2005.

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

[15] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with AMPLE," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pp. 99 – 104, 2005.

[16] J. Clause and A. Orso, "A technique for enabling and supporting debugging of field failures," in *Proceedings of the 29th International Conference on Software Engineering*, pp. 261 – 270, 2007.

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

[18] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273 – 282, 2005.

[19] W. Masri, "Fault localization based on information flow coverage," *Software Testing, Verification and Reliability*, vol. 20, no. 2, pp. 121 – 147, 2010.

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

[21] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[22] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725 – 743, 2007.

[23] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 774 – 787, 1993.

[24] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483 – 1498, 1988.

[25] R. M. Hierons, "Avoiding coincidental correctness in boundary value analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 3, pp. 227 – 241, 2006.

[26] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," in *Proceedings of the 20th international conference on Software engineering*, pp. 188 – 197, 1998.

[27] D. Lorenzoli, L. Mariani, and M. Pezze, "Towards self-protecting enterprise applications," in *Proceedings of the 18th IEEE International Symposium on Software Reliability*, pp. 39 – 48, 2007.

[28] M. J. Harrold and M. L. Soffa, "Efficient computation of interprocedural definition-use chains," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 2, pp. 175 – 204, 1994.

[29] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Transactions on Information and System Security*, vol. 9, no. 1, pp. 61 – 93, 2006.

[30] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 255 – 264, 2002.

[31] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67 – 120, 2012.

[32] C. J. V. Rijsbergen, *Information Retrieval*. Butterworth-Heinemann, 1979.

[33] W. Lee and S. J. Stolfo, *Data mining approaches for intrusion detection*. Defense Technical Information Center, 2000.

[34] W. Masri and A. Podgurski, "An empirical study of the strength of information flows in programs," in *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, pp. 73 – 80, 2006.

[35] G. Liepins and H. Vaccaro, "Anomaly detection: purpose and framework," in *Proceedings of the 12th National Computer Security Conference*, pp. 495 – 504, 1989.

[36] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405 – 435, 2005.

[37] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," *ACM SIGMOD Record*, vol. 22, pp. 207 – 216, 1993.

[38] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proceedings of the 27th international conference on Software engineering*, pp. 432 – 441, 2005.

[39] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, et al., "Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization," in *Proceedings of the 2nd International Conference on Autonomic Computing*, pp. 89 – 100, 2005.

[40] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pp. 2 – 16, 2006.

[41] D. Brumley, H. Wang, S. Jha, and D. Song, "Creating vulnerability signatures using weakest preconditions," in *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pp. 311 – 325, 2007.

[42] A. Chaturvedi, S. Bhatkar, and R. Sekar, “Improving attack detection in host-based IDS by learning properties of system call arguments,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[43] H. Chen, G. Jiang, C. Ungureanu, and K. Yoshihira, “Online tracking of component interactions for failure detection and localization in distributed systems,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 37, no. 4, pp. 644 – 651, 2007.

[44] D. E. Denning, “A lattice model of secure information flow,” *Communications of the ACM*, vol. 19, no. 5, pp. 236 – 243, 1976.

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

[46] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, “Anomaly detection using call stack information,” in *Proceedings of the 2003 Symposium on Security and Privacy*, pp. 62 – 75, 2003.

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

[48] J. T. Giffin, S. Jha, and B. P. Miller, “Automated discovery of mimicry attacks,” *Recent Advances in Intrusion Detection*, pp. 41 – 60, 2006.

[49] D. Jackson and E. J. Rollins, “Chopping: A generalization of slicing,” *technical report*, *DTIC* Document, 1994.

[50] D. Kang, D. Fuller, and V. Honavar, “Learning classifiers for misuse and anomaly detection using a bag of system calls representation,” in *Information Assurance Workshop IAW'05. Proceedings from the 6th Annual IEEE SMC*, pp. 118 – 125, 2005.

[51] H. Kim and B. Karp, “Autograph: Toward automated, distributed worm signature detection,” *USENIX security symposium*, vol. 286, 2004.

[52] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, “On the detection of anomalous system call arguments,” *Computer Security--ESORICS 2003*, pp. 326 – 343, 2003.

[53] 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*, pp. 120 – 132, 1999.

[54] D. Leon, A. Podgurski, and L. J. White, “Multivariate visualization in observation-based testing,” in *Proceedings of the 22nd international conference on Software engineering*, pp. 116 – 125, 2000.

[55] Z. Li, M. Harman, and R. M. Hierons, “Search algorithms for regression test case prioritization,” *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225 – 237, 2007.

[56] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, “Analysis and results of the 1999 darpa off-line intrusion detection evaluation," *Recent Advances in Intrusion Detection*, pp. 162 – 182, 2000.

[57] H. Mannila, H. Toivonen, and A. I. Verkamo, “Discovering frequent episodes in sequences extended abstract,” in *Proceedings of the 1st Conference on Knowledge Discovery and Data Mining*, 1995.

[58] M. Martin, B. Livshits, and M. S. Lam, “Finding application errors and security flaws using pql: a program query language,” *ACM SIGPLAN Notices*, vol. 40, pp. 365 – 383, 2005.

[59] S. McConnell. *Code complete*. O’Reilly Media, Inc., 2004.

[60] W. Masri, “Exploiting the empirical characteristics of program dependences for improved forward computation of dynamic slices,” *Empirical Software Engineering*, vol. 13, no. 4, pp. 369 – 399, 2008.

[61] W. Masri, R. Abou Assi, M. El-Ghali, and N. Al-Fatairi, “An empirical study of the factors that reduce the effectiveness of coverage-based fault localization,” in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, pp. 1 – 5, 2009.

[62] W. Masri and M. El-Ghali, “Test case filtering and prioritization based on coverage of combinations of program elements," in *Proceedings of the 7th International Workshop on Dynamic Analysis*, pp. 29 – 34, 2009.

[63] W. Masri and H. Halabi, “An algorithm for capturing variables dependences in test suites,” *Journal of Systems and Software*, vol. 84, no. 7, pp. 1171 – 1190, 2011.

[64] W. Masri and A. Podgurski, “Application-based anomaly intrusion detection with dynamic information flow analysis,” *Computers & Security*, vol. 27, no. 5, pp. 176 – 187, 2008.

[65] W. Masri and A. Podgurski, “Algorithms and tool support for dynamic information flow analysis,” *Information and Software Technology*, vol. 51, no. 2, pp. 385 – 404, 2009.

[66] W. Masri and A. Podgurski, “Measuring the strength of information flows in programs,” *ACM Transactions on Software Engineering and Methodology*, vol. 19, no. 2, 2009.

[67] W. Masri, A. Podgurski, and D. Leon, “An empirical study of test case filtering techniques based on exercising information flows,” *IEEE Transactions on Software Engineering*, vol. 33, no. 7, pp. 454 – 477, 2007.

[68] F. P. Miller, A. F. Vandome, and J. McBrewster, “Abandonware: Computer software, copyright, office suite, public domain, list of commercial video games released as freeware, orphan works,” 2009.

[69] J. Newsome, D. Brumley, and D. Song, “Vulnerability-specific execution filtering for exploit prevention on commodity software,” in *Proceedings of the 13th Symposium on Network and Distributed System Security*, 2006.

[70] J. Newsome and D. Song, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” in *Proceedings of the 12$^{th}$ Annual Network and Distributed System Security Symposium*, 2005.

[71] A. Nusayr and J. Cook, “Using AOP for detailed runtime monitoring instrumentation,” in *Proceedings of the 7$^{th}$ International Workshop on Dynamic Analysis*, pp. 8 – 14, 2009.

[72] L. Portnoy, E. Eskin, and S. Stolfo, “Intrusion detection with unlabeled data using clustering,” *ACM CSS Workshop on Data Mining Applied to Security*, 2001.

[73] W. Masri and R. Abou Assi, “Cleansing test suites from coincidental correctness to enhance fault-localization,” in *Proceedings of the 3$^{rd}$ IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 165 – 174, 2010.

[74] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5 – 19, 2003.

[75] K. Sen, D. Marinov, and G. Agha, “CUTE: a concolic unit testing engine for C,” in *Proceedings of the 10$^{th}$ European software engineering conference*, 2005.

[76] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, “What we have learned about fighting defects,” in *Proceedings of the 8$^{th}$ IEEE Symposium on Software Metrics*, pp. 249 – 258, 2002.

[77] A. Singh and A. K. Gupta, “A hybrid heuristic for the maximum clique problem,” *Journal of Heuristics*, vol. 12, pp. 5 – 22, 2006.

[78] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, “jRapture: A capture/replay tool for observation-based testing,” in *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pp. 158 – 167, 2000.

[79] D. Wagner and D. Dean, “Intrusion detection via static analysis,” in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 156 – 168, 2001.

[80] H. Xu, W. Du, and S. J. Chapin, “Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths,” *Recent Advances in Intrusion Detection*, pp. 21 – 38, 2004.

[81] 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 14$^{th}$ ACM conference on Computer and communications security*, pp. 116 – 127, 2007.

[82] P. Ammann, J. Offutt, and H. Huang, “Coverage criteria for logical expressions,” in *Proceedings of the 14$^{th}$ International Symposium on Software Reliability Engineering*, pp. 99 – 107, 2003.

[83] J. Farjo, R. Abou Assi, W. Masri, and F. Zaraket, “Does principal component analysis improve cluster-based analysis?,” in *Proceedings of the 6$^{th}$ IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 400 – 403, 2013.

[84] T. Ball and J. R. Larus, “Efficient path profiling,” in Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, pp. 46 – 57, 1996.

[85] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *Proceedings of the International Conference on Software Maintenance*, pp. 368 – 377, 1998.

[86] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, *Verification methodology manual for SystemVerilog*, Springer, 2006.

[87] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al., *Introduction to algorithms*, vol. 2. MIT press Cambridge, 2001.

[88] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv, “User defined coverage - a tool supported methodology for design verification,” in *Proceedings of the 35th annual Design Automation Conference*, pp. 158 – 163, ACM, 1998.

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

[90] J. A. Jones and M. J. Harrold, “Test-suite reduction and prioritization for modified condition/decision coverage,” *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195 – 209, 2003.

[91] W. Masri, R. Abou Assi, F. Zaraket, and N. Fatairi, “Enhancing fault localization via multivariate visualization,” in *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 737 – 741, 2012.

[92] S. Kullback, *Information theory and statistics*. Courier Dover Publications, 2012.

[93] J. W. Laski and B. Korel, “A data flow oriented program testing strategy,” *IEEE Transactions on Software Engineering*, no. 3, pp. 347 – 354, 1983.

[94] I. Neamtiu, J. S. Foster, and M. Hicks, “Understanding source code evolution using abstract syntax tree matching,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1 – 5, ACM, 2005.

[95] S. Rapps and E. J.Weyuker, “Selecting software test data using data flow information,” *IEEE Transactions on Software Engineering*, no. 4, pp. 367 – 375, 1985.

[96] E. Shaccour, F. Zaraket, and W. Masri, “Coverage specification for test case intent preservation in regression suites,” in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 392 – 395, 2013.

[97] W. Yang, “Identifying syntactic differences between two programs,” *Software: Practice and Experience*, vol. 21, no. 7, pp. 739 – 755, 1991.

[98] C. Csallner and Y. Smaragdakis, "DSD-crasher: a hybrid analysis tool for bug finding," in *Proceedings of the 2006 international symposium on Software testing and analysis*, pp. 245 – 254, 2006.

[99] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*, pp. 89 – 98, 2007.

[100] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pp. 88 – 99, 2009.

[101] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pp. 39 – 46, 2006.

[102] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780 – 1792, 2009.

[103] H. Agrawal, J. Horgan, S. London, and W. Wong, "Fault localization using execution slices and data flow tests," in *Proceedings of IEEE Software Reliability Engineering*, pp. 143 – 151, 1995.

[104] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th international conference on Software engineering*, pp. 82 – 91, 2006.

[105] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209 – 224, 2008.

[106] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 595 – 604, 2002.

[107] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *Proceedings of the 23rd international conference on Software engineering*, pp. 339 – 348, 2001.

[108] M. Harman, A. Lakhotia, and D. Binkley, "Theory and algorithms for slicing unstructured programs," *Information and Software Technology*, vol. 48, no. 7, pp. 549 – 565, 2006.

[109] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 3, 2013.

[110] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, no. 4, pp. 371 – 379, 1982.

[111] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

[112] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 16 – 26, 2007.

[113] B. Korel and S. Yalamanchili, "Forward computation of dynamic program slices," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 66 – 79, 1994.

[114] D. Leon, A. Podgurski, and W. Dickinson, "Visualizing similarity between program executions," in *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pp. 311 – 321, 2005.

[115] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *ACM SIGPLAN Notices*, vol. 38, pp. 141 – 154, 2003.

[116] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, pp. 15 – 26, 2005.

[117] 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.

[118] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 286 – 295, 2005.

[119] C. Liu and J. Han, "Failure proximity: a fault localization-based approach," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 46 – 56, 2006.

[120] B. Marick, "The weak mutation hypothesis," in *Proceedings of the symposium on Testing, analysis, and verification*, pp. 190 – 199, 1991.

[121] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proceedings of the 25th International Conference on Software Engineering*, pp. 465 – 475, 2003.

[122] D. J. Richardson and M. C. Thompson, "An analysis of test data selection criteria using the relay model of fault detection," *IEEE Transactions on Software Engineering*, vol. 19, no. 6, pp. 533 – 553, 1993.

[123] S. S. Singh and N. Chauhan, "K-means v/s k-medoids: A comparative study," in *National Conference on Recent Trends in Engineering & Technology*, 2011.

[124] J. M. Voas and K. W. Miller, "Semantic metrics for software testability," *Journal of Systems and Software*, vol. 20, no. 3, pp. 207 – 216, 1993.

[125] X. Wang, S. Cheung, W. K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 45 – 55, 2009.

[126] M. Weiser, "Program slicing," in Proceedings of the 5th International Conference on Software Engineering, pp. 439 – 449, 1981.

[127] W. E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, pp. 42 – 51, 2008.

[128] L. A. Zadeh, "Fuzzy sets," *Information and control*, vol. 8, no. 3, pp. 338 – 353, 1965.

[129] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *Proceedings of the 23rd International Conference on Machine Learning*, pp. 1105 – 1112, 2006.

[130] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. van Gemund, "Diagnosis of embedded software using program spectra," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pp. 213 – 220, 2007.

[131] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246 – 256, 1990.

[132] D. Binkley and M. Harman, "A survey of empirical results on program slicing," *Advances in Computers*, vol. 62, pp. 105 – 178, 2004.

[133] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 342 – 351, 2005.

[134] D. E. Denning, *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[135] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504 – 513, 1977.

[136] J. S. Fenton, "Memoryless subsystems," *The Computer Journal*, vol. 17, no. 2, pp. 143 – 147, 1974.

[137] W. Masri, A. Podgurski, and D. Leon, "Detecting and debugging insecure information flows," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pp. 198 – 209, 2004.

[138] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 5 – 15, 2007.

[139] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121 – 189, 1995.

[140] W. W. Cohen, "Fast effective rule induction," in *Proceedings of the 12th International Conference on Machine Learning*, pp. 115 – 123, 1995.

[141] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," *ACM SIGPLAN Notices*, vol. 41, pp. 169 – 180, 2006.

[142] I. K. Fodor, "A survey of dimension reduction techniques," 2002.

[143] M. Harman and P. McMinn, “A theoretical and empirical study of search-based testing: Local, global, and hybrid search,” *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226 – 247, 2010.

[144] T. Reps, T. Ball, M. Das, and J. Larus, The use of program profiling for software maintenance with applications to the year 2000 problem. Springer, 1997.

[145] J. Shlens, “A tutorial on principal component analysis,” *Systems Neurobiology Laboratory, University of California at San Diego*, vol. 82, 2005.

[146] W. M. Rand, “Objective criteria for the evaluation of clustering methods,” *Journal of the American Statistical association*, vol. 66, no. 336, pp. 846 – 850, 1971.

[147] R. B. Cattell, “The scree test for the number of factors,” *Multivariate behavioral research*, vol. 1, no. 2, pp. 245 – 276, 1966.

[148] L. Hatcher, A step-by-step approach to using the SAS system for factor analysis and structural equation modeling. Sas Institute, 1994.

[149] H. F. Kaiser, “The application of electronic computers to factor analysis,” *Educational and psychological measurement*, 1960.

[150] L. I. Smith, “A tutorial on principal components analysis,” Cornell University, USA, vol. 51, 2002.

[151] I. Borg and P. J. Groenen, Modern multidimensional scaling: Theory and applications. Springer, 2005.

[152] W. Dickinson, D. Leon, and A. Podgurski, “Pursuing failure: the distribution of program failures in a profile space,” *ACM SIGSOFT Software Engineering Notes*, vol. 26, pp. 246 – 255, 2001.

[153] S. Elbaum, A. G. Malishevsky, and G. Rothermel, “Test case prioritization: A family of empirical studies,” *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159 – 182, 2002.

[154] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang, “Estimation of software reliability by stratified sampling,” *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 3, pp. 263 – 283, 1999.

[155] gcc.gnu.org/onlinedocs/gcc/gcov.html

[156] jakarta.apache.org/bcel

[157] sir.unl.edu

[158] jtidy.sourceforge.net