Abstract

ZHENG, JIANG. In Regression Testing without Code. (Under the direction of Dr. Laurie Williams.)

Software products are often built from commercial-off-the-shelf (COTS) components. When new releases of these components are made available for integration and testing, source code is usually not provided by the COTS vendors. Various regression test selection techniques have been developed and have been shown to be cost effective. However, the majority of these test selection techniques rely on source code for change identification and impact analysis. This dissertation presents a regression test selection (RTS) process called Integrated - Black-box Approach for Component Change Identification (I-BACCI) for COTS-based applications. The I-BACCI process reduces the test suite based upon changes in the binary code of the COTS component using the firewall analysis regression test selection method. This dissertation also presents Pallino, the supporting automation that statically analyzes binary code to identify the code change and the impact of these changes. Based on the output of Pallino and the original test suit, testers can determine the regression test cases needed that execute the application glue code which is affected by the changed areas in the new version of the COTS component.

Five case studies were conducted on ABB internal products written in C/C++ to determine the effectiveness and efficiency of the I-BACCI process. The total size of application and component for each release is about 340~930 KLOC. The results of the case studies indicate this process can reduce the required number of regression test

by as much as 100% if there are a small number of changes in the new component in the best case. Similar to other RTS techniques, when there are many changes in the new component the I-BACCI process suggests a retest-all regression test strategy.

With the help of Pallino, RTS via the I-BACCI process can be completed in about one to two person hours for each release of the case studies. Depending upon the percentage of test cases reduction determined by the I-BACCI process, the total time cost of the whole regression testing process can be reduced to 0.0003% of that by retest-all strategy in the best case. Pallino is extensible and can be modified to support other RTS methods for COTS components. Currently, Pallino works on components in Common Object File Format or Portable Executable formats.

In Regression Testing without Code

By

Jiang Zheng

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

COMPUTER SCIENCE

Raleigh, NC

2007

Approved by:

_____	_____
Dr. Jason A. Osborne	Dr. Mladen A. Vouk
_____	_____
Dr. Tao Xie	Dr. Laurie A. Williams Chair of Advisory committee

Dedication

To my parents, wife and daughter

Biography

Jiang Zheng was born on July 27th, 1977 in Hulin, Heilongjiang province, People's Republic of China. He grew up in Shanghai, the largest city of China. He received his Bachelor of Science degree in Computer Science in 1999 from Fudan University, Shanghai, China, and his Master of Science degree in Computer Science in 2005 from North Carolina State University, Raleigh, North Carolina, U.S.A. He also had four years industrial working experience in software development before he was enrolled in North Carolina State University in 2003.

Jiang joined the Software Engineering Realsearch Group (former Laboratory for Collaborative System Development) under the advisement of Dr. Laurie A. Williams in Fall 2003. His research interests include software testing and reliability, component-based software engineering, software reuse, software quality assurance, and software development processes. He had his summer internship in the US Corporate Research of ABB Inc. (Raleigh, NC) in 2005, 2006, and 2007.

Jiang is a student member of Institute of Electrical and Electronics Engineers (IEEE), Association for Computing Machinery (ACM), and the ACM Special Interest Group on Software Engineering (ACM SIGSOFT).

Acknowledgements

This dissertation would never have been accomplished without the generous support and help of many people over many years. First and foremost, I would like to express the greatest gratitude to my advisor, Dr. Laurie A. Williams. I received the first email from Laurie on a Monday, April 14, 2003 at 10:12 AM ET. She introduced me to the interesting world of software engineering research from then on. I am greatly impressed and influenced by her wisdom, insight, diligence, and tolerance. It is very fortunate and enjoyable for me to be her student. I am forever grateful to Laurie for her tireless efforts and tremendous support during my studies.

Dr. Tao Xie has served as a very active and strict committee member and has been continuously giving me advice on how to improve my work. I also greatly appreciate the support given to me by my two other committee members, Dr. Mladen Vouk and Dr. Jason Osborne. Each of them has been valuable sources of information, comments and guidance over the course of this research.

This research was supported by a research grant from ABB Corporate Research. I must gratefully acknowledge ABB Corporate Research for supporting me financially for the past three years, including my internship during the three summer semesters. Very special thanks to Brian Robinson and Karen Smiley, who have contributed enormously to this research. This dissertation would not have been possible without the excellent collaboration with you.

Many thanks to researchers from all over the world who have commented on my

research in various emails, meetings, conferences and doctoral symposium, in alphabetical order: Dr. Lawrence Chung, Dr. Steve Easterbook, Dr. Xavier Franch, Dr. Rose Gamble, Dr. Jerry Gao, Dr. David Kay, Dr. Forrest Shull, Dr. Ann Sobel, and Dr. Sebastian Uchitel. I am also indebted to Dr. Cem Kaner, a lawyer and software engineering professor, for his patient explanation and immense help on the legal issues of this dissertation, and for allowing me to use his legal writing in Chapter 6. Additionally, special thanks to Dr. Nachiapan Nagappan, Will Snipes, and John Hudepohl for their contribution in the automatic static analysis research and the IEEE Transactions in Software Engineering (TSE) paper.

I have had the fortunate and honor to work with and learn from many faculty members at NC State including: Dr. Annie Antón, Dr. Rada Chirkova, Dr. Do Young Eun, Dr. Thomas Honeycutt, Dr. Purushothaman Iyer, Ms. Carolyn Quarterman, Dr. Injong Rhee, Dr. Carla Savage, Dr. Roger Woodard, Dr. Peter Wurman, and Dr. Ting Yu. A special thanks to Dr. Antón for her great encouragement and advice when I was her teaching assistant in Spring 2004. I also appreciate the financial assistance I have received from the Department of Computer Science at North Carolina State University. Many thanks to Dr. David Thuente, Margery Page, Carol Holloman, Ginny Adams, Susan Peaslee, Linda Honeycutt, Ken Tate, Vilma Berg, Missy Seate, Carol Allen, and Sarah Williams, who have constantly provided assistance and guidance during the past four years.

I would like to take this chance to thank my office mates, both past and present,

in the Software Engineering Realsearch Group for their great encouragement and help. I have benefited greatly from working with an amazing group of students: Michael Gegick, Sarah Heckman, Dright Ho, Neha Katira, Lucas Layman, Andy Meneely, Meiyappan Nagappan, Dr. Nachiappan Nagappan, Mark Sherriff, Yonghee Shin, Ben Smith, Dr. Hema Srikanth, and Stephen Thomas. This dissertation and my other publications would not have been possible without the insightful discussion and comments in our reading groups. I am also grateful to Dr. Qingfeng He for his support on my job hunting. There are numerous other friends both inside and outside NC State University that have been helpful over the years and their assistance is greatly appreciated.

Finally, but definitely not least, I am especially grateful to my family for their unconditional love, support, encouragement, and patience during my graduate studies. My parents, Jingzi Zhou and Dali Zheng, have been constant sources of love and support for which I will forever be thankful. My beloved wife, Yushen Huang, has been accompanying me on the trip to my Ph.D. far away from her parents, and very supportive of both my living and study. Thanks also go to my adorable little princess, Grace Xinyue Zheng, whose brilliant smiles every morning have been encouraging me for nine months.

Table of Contents

List of Tables

List of Figures

Glossary

Affected exported component functions: *Affected exported component functions* are functions within the COTS component that interface with the application, and are either changed or are in the call chain of other changed functions.

Affected component functions: *Affected component functions* are functions within the COTS component that are in the call chain of the changed or added component functions.

Affected glue code functions: *Affected glue code functions* are functions within the glue code that directly call affected exported component functions.

American Law Institute (ALI): *The American Law Institute* was established in 1923 to promote the clarification and simplification of American common law and its adaptation to changing social needs. The ALI drafts, approves, and publishes restatements of the law, model codes, and other proposals for law reform. [1]

Binary code analysis (BCA): Binary code analysis is the activity that analyzes the binary executable code of software to facilitate many software development-related activities, including program comprehension, software maintenance and software security.

Black-box testing: *Black-box testing*, also called *functional testing* or *behavioral testing*, is testing that ignores the internal mechanisms of a system or component and focuses solely on the outputs generated in response to selected inputs and execution

conditions [38].

Calling virtual address: For each key-value pair in the relocation index, the key is a *calling virtual address* where the control flow jumps to another function or data.

Common Object File Format (COFF): The *Common Object File Format* is a specification of a format for executable, object code, and shared library computer files. COFF libraries usually have the extension .lib [55].

Controlled Regression Testing: *Controlled Regression Testing* assumes that factors other than the program (such as the operating environment, the nondeterministic ordering of statements in concurrent programs, or databases and files that contribute data) do not affect test execution [63].

Decompose: The term *decomposing* is used to refer to breaking up the binary code down into constituent elements, such as code sections and relocation tables.

Dynamic binary code analysis: *Dynamic binary code analysis* monitors the execution of programs.

End virtual address: *End virtual address* is one of the main attributes in the function/data model indicating the virtual address of the last byte of a functions or data.

Explanation type theory: An *explanation type theory*, also labeled as "theory for understanding," provides explanations but does not aim to predict with any precision, and there are no testable propositions [30].

Fair use: American courts have repeatedly ruled that software reverse

engineering is perfectly legal because the Copyright Act allows certain types of copying without permission. Collectively, these are called *fair use*. Photocopying part of a book for classroom teaching is an example of fair use [39].

Fault-revealing test cases: *Fault-revealing test cases* are those test cases detect one or more faults in P' if it causes P' to fail [63].

Firewall analysis: *Firewall analysis* is a regression test selection technique for regression testing with integration test cases in the presence of small changes in functionally-designed software [80].

Glue code: *Glue code* is application code that interfaces with the COTS components, integrating the component with the application.

Magic number: A *magic number* is a pre-defined constant, typically located at the first few bytes of a binary file, used to identify the file type [52].

Model-view-controller (MVC): An model-view-controller architecture separates data (model) and user interface (view) concerns, so that changes to the user interface do not affect the data handling, and that the data can be reorganized without changing the user interface.

Modification-traversing test cases: *Modification-traversing test cases* for P and P' are those test cases executing new or modified code in P', or formerly executed code that has since been deleted from P [63].

Modification-revealing test cases: *Modification-revealing test cases* are those test cases, when executed before and after the modification, the program will generate

different output [63].

The National Conference of Commissioners on Uniform State Laws (NCCUSL): *The National Conference of Commissioners on Uniform State Laws* is a non-profit, unincorporated association in the United States that consists of commissioners appointed by each state and territory. The purpose of the association is to discuss and debate in which areas of law there should be uniformity among the states and to draft acts accordingly. [2]

Notation for regression testing: Let P be a program and P' be a modified version of P. Let S and S' be the specifications for P and P', respectively. Let T be a test suite developed initially for P [63].

Obsolete test cases: *Obsolete test cases* for P' are those test cases if and only if the expected output of t has changed for S' [63].

Portable Executable (PE): The *Portable Executable* format is a file format for executables, object code, and DLLs, used in Windows operating systems. Typical PE files have the extensions .exe, .dll, .ocx, .sys, .cpl and .scr. [52]

Regression testing: *Regression testing* involves selective re-testing of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [38].

Regression test selection (RTS): *Regression test selection* techniques attempt to reduce the time required to retest a modified program by selecting some subset of the existing test suite. The purpose of RTS techniques is to reduce the high cost of

retest-all regression testing by selecting a subset of possible test cases. [63]

Relative Virtual Address (RVA): A *relative virtual address* is the virtual address of an object from the file once it is loaded into memory, minus the base address of the file image.

Relocation index: In the algorithm of analyzing PE components, the relocation table is read from the relocation section and then converted into a `Hashtable` called *relocation index.*

Relocation table set: The algorithm of analyzing COFF components collects and saves the relocation tables of the functions into a text file called "relocation table set".

Reverse engineering by the Bowers v. Baystate Technologies case: *Reverse engineer* is to study or analyze (a device, as a microchip for computers) to learn details of design, construction, and operation, perhaps to produce a copy or an improved version [3, p. 1326].

Software change impact analysis: *Software change impact analysis*, or *impact analysis* for short, estimates what will be affected in software and related documentation if a proposed software change is made [7].

Start virtual address: *Start virtual address* is one of the main attributes in the function/data model indicating the virtual address of the first byte of a functions or data.

Static binary code analysis: *Static binary code analysis* provides a way to obtain information about the possible states that a program reaches during execution without

actually running the program on specific inputs.

Reliable: Test cases are *reliable* if the correctness of modules exercised by those tests for the tested inputs implies correctness of those modules for all inputs [63].

Safe RTS technique: A *safe RTS technique* guarantees that the subset of tests selected contains all test cases in the original test suite that can reveal faults based upon the modified program [63].

Target virtual address: For each key-value pair in the relocation index, the start virtual address of the function or data being called (a.k.a. *target virtual address*) is stored as the value.

Virtual address: A *virtual address* is an address identifying a virtual (non-physical) entity.

What You See Is Not What You eXecute (WYSINWYX) phenomenon: There can be a mismatch between what a programmer intends and what is actually executed by the processor, e.g., presence or absence of procedure calls by the optimizing compiler [9].

CHAPTER 1

INTRODUCTION

Companies increasingly incorporate a variety of commercial-off-the-shelf (COTS) components in their products. Upon receiving a new release of a COTS component, users of the component often conduct regression testing to determine if the new version of the component will cause problems with their existing software and/or hardware system. *Regression testing* involves selective re-testing of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [38]. A variety of regression test selection (RTS) processes have been developed [5, 10-12, 14-16, 21, 22, 26, 29, 32-37, 42, 44, 47-49, 51, 57, 63-69, 72-74, 80, 82, 84] to reduce the number of tests that need to be executed without significant risk of excluding important failure-revealing test cases. However, most existing RTS techniques rely on source code, and therefore are not suitable when source code is not available for analysis, such as when an application incorporates COTS components.

Due to the lack of information, the most straightforward RTS strategy for COTS-based applications would be to rerun all of the test cases for the application involving the glue code after the new COTS component(s) have been integrated.

Glue code is application code that interfaces with the COTS components, integrating the component with the application. The retest-all strategy can be prohibitively expensive in both time and resources [26]. The goal of our research is to*, with minimal reduction in regression fault detection ability, reduce the regression testing required for COTS-based applications when components change and source code is not available*.

To advance the body of knowledge, theory should be an integral part of empirical studies in software engineering [30]. Empirically-based theories are generally perceived as foundational to science [30]. An explanation type theory will be built through this research. An *explanation type theory*, also labeled as "theory for understanding," provides explanations but does not aim to predict with any precision, and there are no testable propositions. These theories often have an emphasis on showing others how the world may be viewed in a certain way, with the aim of bringing about an altered understanding of how things are or why they are as they are. [27, 30]

The explanation type theory being built via this research is stated as follows:

When components change and source code is not available, regression tests can be selected from the test cases that execute the glue code that is in the call chain of functions of the component that changed, with minimal reduction in regression fault detection ability.

A multi-step RTS process with supporting automation has been evolved for

COTS-based applications [85-90]. The process is called *the Integrated - Black-box Approach for Component Change Identification (I-BACCI[1])* process. *Black-box testing*, also called *functional testing* or *behavioral testing*, is testing that ignores the internal mechanisms of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions [38]. The I-BACCI process is an integration of (1) a static binary code analysis for change identification and impact analysis; and (2) the firewall analysis RTS technique. Our uniqueness is the combination of the two parts to identify and localize change with the goal of reducing the regression test suite.

The input artifacts to the process are (1) the binary code of the components (old and new versions); (2) the source code of the glue code of the user application not including the COTS components; and (3) all test cases which are mapped to the glue code functions they execute. These input artifacts are generally available to users of COTS components. The output of the I-BACCI process is a reduced suite of regression test cases that execute the application glue code that is in the call chain of changed functions in the new COTS components.

The I-BACCI process has evolved to Version 4 through the application of the process on components in Common Object File Format (COFF) [55] and the Portable Executable (PE) format [58, 59] written in C/C++. COFF libraries usually have the extension `.lib`. Typical PE files have the

[1] We pronounce BACCI the way the bocce is pronounced when referring to the Italian ball game: [bah-chee].

extensions `.exe`, `.dll`, `.ocx`, `.sys`, `.cpl` and `.scr`. Additionally, the comprehensive open source supporting automation for the I-BACCI process, henceforth called *Pallino*[2], has been developed to perform static binary change identification and impact analysis. Pallino outputs a list of affected exported component functions. *Affected exported component functions* are functions within the COTS component that interface with the application, and are either changed or are in the call chain of other changed functions. Based on the list of affected exported component functions and the original test suite, testers can identify glue code functions that call affected exported component functions. Then, the subset of the regression test cases that execute the glue code which is affected by the changed areas in the new COTS components can be identified. Currently Pallino works on components in COFF or PE format written in C/C++ and functionality that processes other binary file format can be easily added into the tool without affecting many other modules. Pallino can be modified to support other RTS methods for COTS components.

This paper reports the results of applying the I-BACCI Version 4 process to identify a reduced test suite for four industrial case studies. Two of the case studies involve COFF component, and two involves PE component. These results are used to analyze the effectiveness and efficiency of the I-BACCI as an RTS process. Also, another case study was conducted to evaluate the overall efficiency of the firewall analysis RTS technique that is used in the I-BACCI process.

[2] A pallino is the small ball used in the bocce ball game. http://www4.ncsu.edu/~jzheng4/pallino.htm

The rest of this dissertation is organized as follows. Chapter 2 outlines the background and related work. Chapter 3 describes the I-BACCI Version 4 process and its limitations. Supporting automation is discussed in Chapter 4. Chapter 5 presents the case studies of applying I-BACCI Version 4 on ABB products and their components. Related legal issues are discussed in Chapter 6. Finally, Chapter 7 presents the conclusions and contributions. The summary of binary code comparison false positive patterns is presented in Appendix A.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides information about prior work in testing of software components, regression test selection, firewall analysis, binary code analysis, change identification, and impact analysis.

2.1 TESTING OF SOFTWARE COMPONENTS

Poor testability, due to the lack of access to the component's source code and other artifacts, is one of the challenges in user-oriented component testing [24, 25, 76]. The major objectives of user-oriented component testing are as follows [24]:

(1) to validate the functions and performance of a reusable component relative to its specifications;

(2) to confirm the proper usage and deployment of a reusable component in a specific platform and operation environment; and

(3) to check the quality of customized components developed using reused components.

Because a third-party component user only has access to the component specification, user interfaces, and reference manual, the challenges in user-oriented

component testing include [25]:

(1) difficulties understanding the functions and behaviors of reusable components;

(2) inability to perform white-box program analysis to support debugging; and

(3) difficulties conducting unit-level black-box testing without access to the component creators and related artifacts.

Generally, black-box tests are run on COTS software because users do not have access to the source code to analyze the internal implementation. Black-box test cases of COTS component functionality can be based upon the specification documentation provided by the vendor. Alternately, the behavior could be determined by studying the inputs and the related outputs of the component. When only binary code is available, binary reverse engineering is a technically-feasible approach for automatically deriving information that can inform the RTS. The derived information can be a program structure of a component from its binary code, such as, call graphs [52].

Harrold et al. [31] presented techniques that use component metadata for regression test selection of COTS components. They illustrated their technique with a controlled example and seven releases of a real component-based system, demonstrating an average savings of 26% of the testing effort [31]. Their techniques utilize three types of metadata to perform the regression test selection: (1) the branch coverage achieved by the test suite with respect to the component to associate test cases with branches; (2) the component version; and (3) a way to query the

component for the branches affected by changes in the component between two given versions [31]. However, the component vendors may not provide the metadata information, especially the third type of metadata. Also, the cost of providing the metadata may be high in practice. [31] In this research, we focus on using information that is typically available to a COTS component user.

2.2 REGRESSION TEST SELECTION

The purpose of RTS techniques is to reduce the high cost of retest-all regression testing by selecting a subset of possible test cases [26]. A variety of RTS techniques [5, 10-12, 14-16, 21, 22, 26, 29, 32-37, 42, 44, 47-49, 51, 57, 63-69, 72-74, 80, 82, 84] have been proposed, such as methods based upon path analysis techniques or dataflow techniques. Rothermel and Harrold evaluated and compared the RTS techniques by establishing a framework composed of four categories: inclusiveness, precision, efficiency, and generality [63]:

- *Inclusiveness* measures the ability of a technique to choose tests that will cause the modified program to produce different output than the original program, and thereby expose faults caused by modifications.
- *Precision* measures the ability of a technique to eliminate or exclude tests that will not cause the modified program to produce different output than the original program. No techniques are 100% precise.
- *Efficiency* measures the computational cost of a technique.
- *Generality* measures the ability of a technique to handle realistic and diverse

language constructs, arbitrarily complex code modifications, and realistic testing applications. [63]

Rothermel and Harrold also formally defined the notation and terms for regression testing [63]. Let P be a program and P' be a modified version of P. Let S and S' be the specifications for P and P', respectively. Let T be a test suite developed initially for P. *Fault-revealing test cases* are those test cases detect one or more faults in P' if it causes P' to fail [63]. A *safe* RTS technique guarantees that the subset of tests selected contains all test cases in the original test suite that can reveal faults based upon the modified program [14, 45, 63, 64]. Generally, a superset of the set of test cases in T that are fault-revealing for P' can be selected under certain conditions, although there is no effective algorithm for finding the test cases in T that are fault-revealing for P' [62, 63]. *Modification-revealing test cases* are those test cases, when executed before and after the modification, the program will generate different output [63]. The modification-revealing test cases in T form a set equivalent to the set of fault-revealing test cases in T, given two assumptions that (1) When P was tested with t, P halted and produced correct output, for each test case t belonging to T; and (2) for each test case t belonging to T, t is not obsolete for P'. *Obsolete* test cases for P' are those test cases if and only if the expected output of t has changed for S'. [63]

When an update of a COTS component replaces the old version of the component, the specification of users' application that incorporates the component often does not

change. We may only consider the non-obsolete test cases in *T* for users' application in user-oriented component testing. However, there is no effective algorithm for precisely identifying the non-obsolete test cases in *T* that are modification-revealing for *P* and *P′*, even when the above two assumptions hold [62, 63]. *Modification-traversing test cases* for *P* and *P′* are those test cases executing new or modified code in *P′*, or formerly executed code that has since been deleted from *P*. A non-obsolete test case can only be modification-revealing for *P* and *P′* if it is modification-traversing for *P* and *P′* when a third assumption (Controlled Regression Testing) holds, with the above two assumptions. *Controlled Regression Testing* assumes that factors other than the program (such as the operating environment, the nondeterministic ordering of statements in concurrent programs, or databases and files that contribute data) do not affect test execution. [63] The relationship that holds among these terms for non-obsolete test cases in *T* when the assumptions hold is show in Figure 2.1. The set of modification-traversing test cases is a superset of the set of modification-revealing test cases.

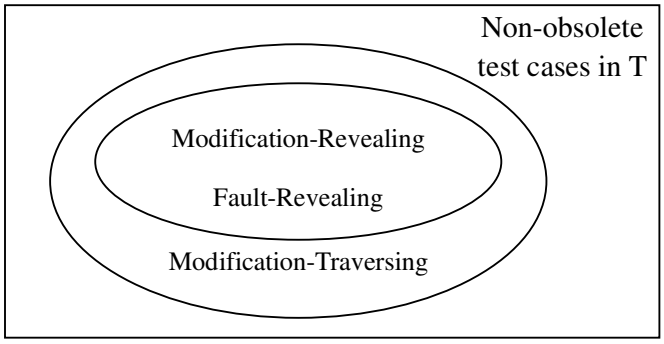


Figure 2.1: Relationship among classes of test cases for non-obsolete test cases, adapted from [63]

Rothermel and Harrold analyzed RTS techniques and determined that four RTS techniques are safe for controlled regression testing [63]:

- linear equation (nonminimization) [22],
- cluster identification [42],
- modified entity [16], and
- graph walk [62, 64-66].

Fischer presents the linear equation, a selective retest technique that uses systems of linear equations to select test suites that yield segment coverage of modified code. Linear equation techniques use systems of linear equations to express relationships between tests and program segments. The analysis time cost in the worst case is exponential in the program size, which may be very expensive in practice [22, 63]. The techniques may be data and computation intensive on large programs due to the calculations required to solve systems of linear equations [22, 63].

Laski and Szermer [42] present the cluster identification, a technique for identifying single-entry, single-exit subgraphs of a control flow graph (CFG), called clusters, that have been modified from one version of a program to the next. The cluster identification technique computes control dependence information for a procedure and its changed version, and then computes the control scope of each decision statement in the procedure by taking the transitive closure of the control dependence relation. The cluster identification technique is an intra-procedural technique and does not support inter-procedural regression testing [42]. Also, the

cluster identification technique selects non-modification-traversing tests and is less precise than the graph walk technique [63].

Chen, Rosenblum, and Vo [16] present the modified entity technique, a regression test selection technique that detects modified code entities. Code entities are defined as executable portions of code such as functions, or as nonexecutable components such as storage locations. First, a program is instrumented to generate traces of all functions called during its execution. Execution of a test case on the instrumented system generates a trace of the functions. Also, all of the (static) references between the entities in the source code of the program are stored in a database. When a new version is created, a source code database is created for the new version. The databases for both old and new version of a program are compared to identify the entities that were changed. Then the technique compares the list of changed entities with the coverage information preserved for each test case, and selects all test cases that cover changed entities for retesting the new version of the program. [16] Because the modified entity technique may also select tests that do not execute modified clusters or modified execution traces, the modified entity technique is less precise than the cluster identification or CFG-walk techniques [14, 63]. Although it is the most efficient safe RTS technique, the modified entity technique would still take more time executing the selected tests than that of cluster identification and graph walk techniques [63]. Also, the technique requires the use of a database containing information about code [16, 63].

Rothermel and Harrold presented the graph walk technique. Graph walk involves the following five steps: (1) construct the CFGs for a procedure or program and its modified version; (2) collect traces for tests in the original test suite that associate tests with CFG edges; (3) perform synchronous depth-first traversals of the two graphs; (4) comparing the program statements associated with those nodes that are reached along prefixes of execution traces; and (5) use these graphs to select tests that execute changed code from the original test suite [63, 64]. The graph walk techniques select all modification-traversing tests, a superset of fault-revealing test cases and modification-revealing test cases, and are safe for controlled regression testing. Also, the graph walk technique is the most precise safe technique by selecting tests through modified programs at a finer grain [63].

Graph walk techniques gain their safeness and precision by increasing the costs of analysis [63]. For example, a mapping between each test case and CFG edges traversed by the test case must be maintained [64]. Especially for large scale industrial software project, it is very complex to analyze the control flow graph and execution traces in such a fine granularity for the whole system. Rothermel and Harrold suggest that it is helpful to conduct empirical studies to determine whether a safe RTS technique, such as the modified entity technique and graph walk technique, offers sufficient improvements in fault detection in comparison to a nonsafe but efficient RTS technique, such as the firewall technique [63]. A case study that compares the total time costs for regression testing (i.e., the overall efficiency) of firewall analysis

with those of modified entity technique, the most efficient safe RTS technique, will be discussed in Chapter 5.6.

Srivastava and Thiagarajan at Microsoft developed Echelon [71], a test prioritization system. Echelon is used to prioritize tests based upon changes between two versions identified by a binary code comparison. Echelon takes as input two versions of the program in binary form, and a mapping between the test suite and the lines of code it executes. Echelon outputs a prioritized list of test sequences (small groups of tests). The researchers analyzed the efficacy of Echelon based on two runs of a comparison between two binaries of a 1.8 million line of code office productivity application [71]. In the first run, Echelon detected 87% of the defects in the first two of 148 test sequences; the remaining 13% of the defects were not detected by any tests in the test suite. In the second run of different binaries, Echelon detected 98% of the defects in the first three of 221 test sequences; the remaining 2% of the defects were not detected by any tests.

However, Echelon is a large proprietary Microsoft internal product with a significant infrastructure and an underlying binary code manipulation engine, and therefore cannot be used by the community at large. Also, Echelon prioritizes, but does not eliminate tests [71]. Our goal is to provide information about which test cases are not necessary to rerun.

Mariani et al. [49] proposed (1) a technique for automatically deriving small compatibility test suites to evaluate several alternative candidate components that can

replace a COTS component within a software system; and (2) a technique for prioritizing test cases to improve the efficiency of regression testing of COTS components, when integrated in new software systems to update obsolete components. Their techniques derive information from software characteristics by focusing on behavioral models that are automatically extracted from the execution of test cases on previous versions of the software, and by computing priorities according to the exercised behaviors instead of the specific changes. Their techniques rely on automatic inference of two types of behavioral models: input/output (I/O) and interaction models. *I/O models* are boolean expressions over the values exchanged during the computation, describing properties of data values exchanged between components, and are inferred from traces with the Daikon engine [20]. *Interaction models* describe sequences of invocations by finite state machines labeled with method invocations, and are derived with the kBehavior engine [50]. Regression test cases are prioritized according to the complexity of the interactions between the system and the target component that are triggered by the test, because "during integration testing, long interactions are more likely to expose faults than simple ones, which should be already covered by the unit testing of the component." Their empirical investigations conducted so far showed that (1) the generated compatibility test suites were less than 7% of the original test suites, and could identify incompatible components within a set of syntactically-compatible candidates, and reveal about 77% of the integration faults; and (2) prioritized test cases could reveal

96% of faults while executing less than 10% of test cases. [49] As a dynamic analysis technique that automatically synthesizes behavioral models from execution traces, their techniques require runtime setup and reliable sets of test suites. Also, their techniques prioritize instead of eliminating test cases. Our approach selects test cases needed to rerun based on static binary code analysis which will be discussed in Chapter 2.4.

2.3 FIREWALL ANALYSIS

Leung and White [4, 45, 46, 79] developed firewall analysis for regression testing with integration test cases (tests that evaluate interactions among components [38]) in the presence of small changes in functionally-designed software. Module dependencies, control-flow dependencies, and data dependencies are considered in firewall analysis [79]. Dependencies are modeled as call graphs. Firewall analysis restricts regression testing to potentially-affected system elements that directly call changed system elements, i.e., system elements that can be reached within one edge from changed system elements in the call graph [79, 81]. Affected system elements include: modified functions and data structures, and their calling functions. A "firewall" is “drawn” around the changed functions on the call graph. All modules inside the firewall are unit and integration tested, and are integration tested with all modules not enclosed by the firewall [79]. Test cases that need to be re-run over these modules are identified and/or new test cases to exercise new code or functionality are generated. For example, as illustrated in the call graph of system

elements in Figure 2.2, system elements 1, 2, and 3 are modified, and the testing firewall requires that unit and integration tests be conducted for system elements 1-8. Kung et al. [40, 41] utilized the firewall concept on an object-oriented system, and White and Abdullah [77] expanded the firewall to address more features of an object-oriented system. Firewall has also been utilized in the regression testing of graphical user interfaces [78].

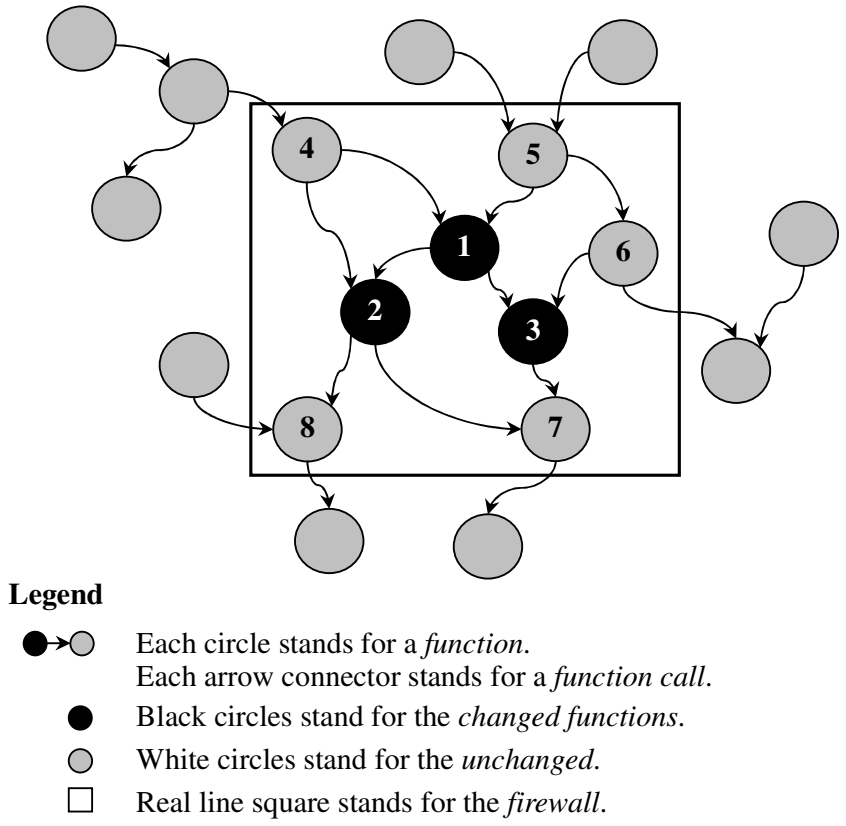


Figure 2.2: Illustration of firewall analysis

Firewall methods can only be guaranteed to select all modification-revealing [63] tests and to be safe if all unit and integration tests initially used to test system components are reliable. Tests are *reliable* if the correctness of modules exercised by

those tests for the tested inputs implies correctness of those modules for all inputs [63]. Unfortunately, test suites are typically not reliable in practice [81], so the firewall technique may omit modification-revealing tests and/or may admit some non-modification-traversing tests. When test suites are not reliable, there may exist some input *i* that belongs to the input domain of a modified function *p*, such that no test in the test suite selected by firewall analysis exercises *p* with input *i*, and such that input i may expose a fault in *p*. If some system test *t* is within the original test suites but not within the selected test suite, and such that *t* causes *p* to be invoked with the input *i*, then *t* is fault-revealing, but the firewall technique does not select *t*. [63] However, via empirical studies of industrial real-time systems, firewall analysis was shown to be effective [81]. The firewall analysis allowed an average savings of 36% of the testing time and 42% of the number of the tests run. No additional errors were detected by the customer on the studied software releases that were due to the changes in these releases to date. There were additional defects found, but none were regression or unit test failures. These were pre-existing defects that were created in the original release. [81].

Compared to the graph walk techniques, firewall analysis takes the advantage of the simplicity and efficiency in change impact and test selection analysis, especially for large scale complex industrial software projects with integration test cases [63, 64, 80, 81]. Firewall analysis is more efficient and precise than the modified entity by analyzing less entities in the programs and selecting less test cases, and therefore

spends less time on rerunning selected test cases, especially for regression testing components in very deep levels of the call chains in large scale complex industrial software projects [16, 63, 80]. Firewall is the RTS technique embodied in the I-BACCI Version 4. Based upon the results of these empirical studies of firewall, these theoretical limitations of firewall should not impair the effectiveness of the I-BACCI process in practice. As will be discussed in Chapter 3, our approach extends the traditional concept and scope of firewall analysis for use with binary code.

2.4 BINARY CODE ANALYSIS

Software developers generally write source code in high level programming languages. Compilers translate that source into binary code that computer processors can recognize and execute. During maintenance, the development team analyzes the source or binary code to determine the course of their corrective action. For example, developers can use their knowledge of what was changed at the function/method and line of code level to determine what to retest before releasing the evolved software to customers. The developers have free reign to analyze their source manually or with tools, to test, integrate or revise the code.

Software purchasers, however, may not have access to the source code. Binary code analysis (BCA) approaches and tools have been developed and utilized in many software development-related activities, including program comprehension, software maintenance and software security, even by software developers that have access to the source code [71]. For example, malicious code or patterns in executable can be

detected via BCA to enhance software security [13].

Balakrishnan et al. presented the "What You See Is Not What You eXecute" (WYSINWYX) phenomenon: There can be a mismatch between what a programmer intends and what is actually executed by the processor, e.g., presence or absence of procedure calls by the optimizing compiler [9]. Therefore, analyses performed on source code can fail to detect certain bugs and vulnerabilities. Also, analyzing executables has other advantages, such as, revealing more accurate information about the behaviors that might occur during execution, because an executable contains the actual instructions that will be executed [9].

Additionally, BCA can be dynamic or static. Dynamic BCA monitors the execution of programs. In contrast, static BCA provides a way to obtain information about the possible states that a program reaches during execution without actually running the program on specific inputs. Static techniques explore the program's behavior for all possible inputs and all possible states that the program can reach. [9] Srivastava and Thiagarajan discussed the advantages of comparing software at the binary level rather than the source code level [71]:

(1) easier to integrate into the build process because the recompilation step needed to collect coverage data is eliminated; and

(2) all the changes in header files (such as constants and macro definitions) have been propagated to the affected procedures, simplifying the determination of program changes.

There are many examples of static BCA. To obtain the strings that contain significant high-level information about the program and its communication with the runtime environment, Christodorescu et al. [18] presented a static analysis technique for recovering possible string values in an executable program when no debug information or source code is available. Malicious code or patterns in executable programs can be statically detected [13]. CodeSurfer uses a static-analysis algorithm called value-set analysis (VSA) to recover intermediate representations similar to those a compiler creates for a program written in a high-level language [8]. Our tool utilizes static BCA to identify changes and change impact within the COTS components where source code is not available.

2.5 CHANGE IDENTIFICATION AND IMPACT ANALYSIS

A key step in choosing regression tests is applying impact analysis [56] to identify changes between the new release and the previously-tested version with the same source code base. *Software change impact analysis*, or *impact analysis* for short, estimates what will be affected in software and related documentation if a proposed software change is made [7]. However, similar to RTS, most change identification and impact analysis approaches utilize the source code of the old and modified programs [6, 42, 60, 61, 63, 73, 74]. These approaches are not suitable for component testing when source code is not available.

Laski and Szermer [42] proposed a formal method to identify modifications made in a program. Vokolos and Frankl [73, 74] utilized a textual differencing technique to

perform regression test selection. Apiwattanapong et al. [6] presented a technique for comparing object-oriented programs that identifies both differences and correspondences between two versions of a program. The algorithm is based on a method-level representation that models the object-oriented features of the language. Given two programs, their algorithm identifies matching classes and methods, builds a representation for each pair of matching methods, and compares the representation to identify similarities and differences. Empirical results show that the technique achieves improvements from 17% to over 65% in terms of matching unchanged parts of the code, compared to Laski and Szermer's algorithm [6]. Ren et al. [60] developed Chianti, a change impact analysis tool for Java. Chianti analyzes two versions of a Java program and decomposes their difference into a set of atomic changes. Change impact is then reported in terms of affected (regression or unit) tests whose execution behavior may have been modified by the applied changes. For each affected test, Chianti also determines a set of affecting changes that were responsible for the test's modified behavior [60, 61]. Their empirical results show that after a program edit, on average the set of affected tests is 62.4% of all the possible tests and for each affected test, the number of affecting changes is very small (5.9% of all atomic changes in that edit) [61].

Wang et al. [75] developed the Binary Matching Tool (BMAT) which compares two versions of a binary program without knowledge of the source code changes. The implementation uses a hashing-based algorithm and a series of heuristic methods to

find correct matches for as many program blocks as possible. The algorithm first matches procedures, then basic blocks within each procedure. The implementation of BMAT is built on Windows NT® for the x86 architecture, using the Vulcan binary analysis tool [70] to create an intermediate representation of x86 binaries. Vulcan separates code from data and identifying program symbols. The process enables good matching even with shifted addresses, different register allocations, and small program modifications [75]. BMAT underlies Echelon [71] (discussed in Chapter 2.2) to match blocks in the two binaries. However, like Echelon, BMAT is a proprietary tool. We have developed a lightweight non-proprietary tool to perform the similar function for the I-BACCI process.

CHAPTER 3

THE I-BACCI PROCESS

This chapter presents the I-BACCI RTS process. Chapter 3.1 presents the high level information including the evolution of the process. Chapter 3.2 introduces the detailed steps of the process. An illustrative example is presented in Chapter 3.3. The limitations of the process is discussed Chapter 3.4.

3.1 INTRODUCTION AND PROCESS EVOLUTION

The I-BACCI process is an integration of (1) a static BCA for change identification and impact analysis; and (2) the firewall analysis RTS technique. Our uniqueness is the combination of these two parts to identify and localize changes with the goal of reducing the regression test suite. The I-BACCI Version 4 involves seven steps, as shown in Figure 3.1. The first four steps are completed via a BCA process (in dash-dotted line frame) using the Pallino tool, which produces a report on affected exported component functions. The remaining three RTS steps are completed via firewall analysis (in dashed line frame) and ultimately produce the reduced set of test cases that need to be rerun.

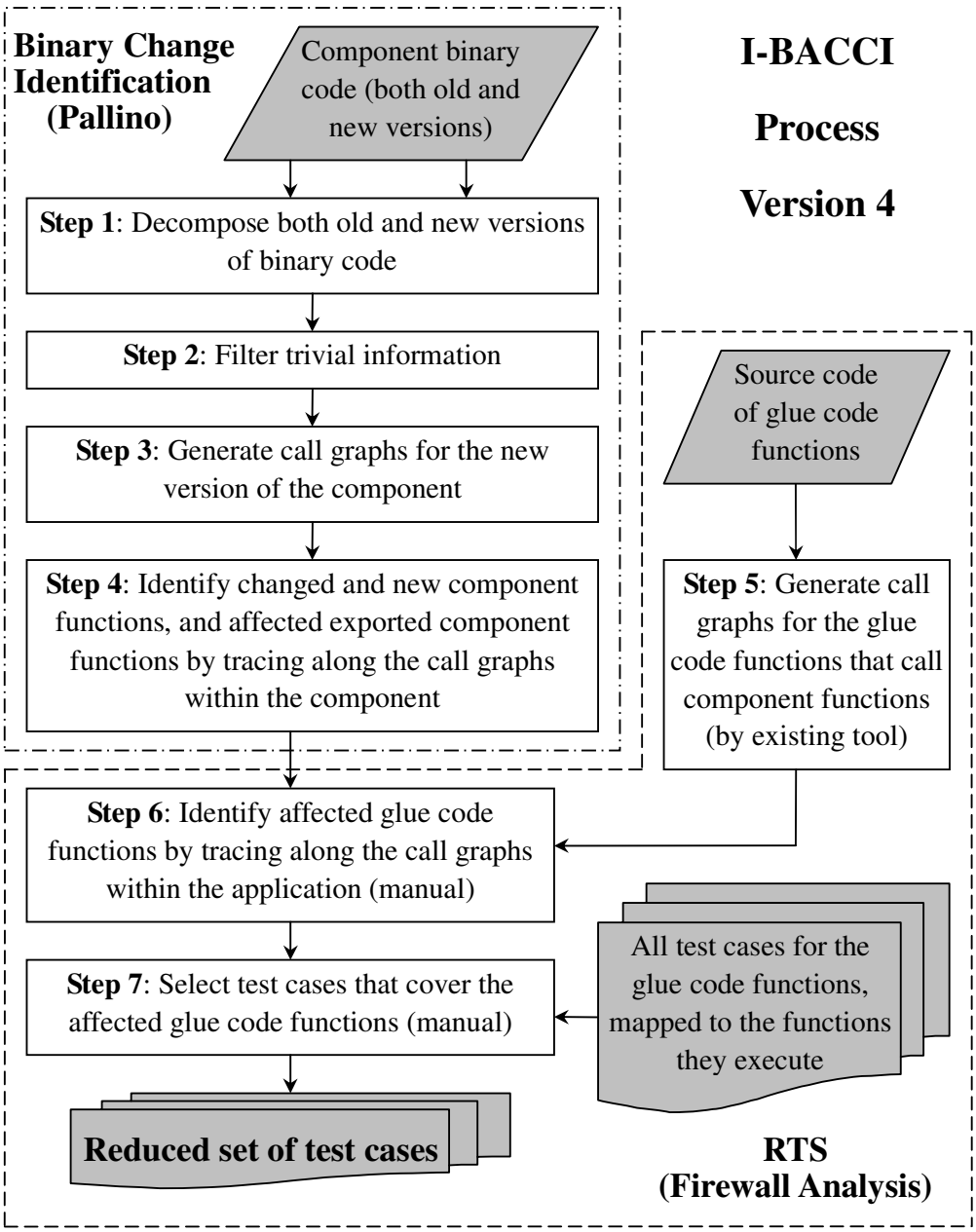


Figure 3.1: The I-BACCI version 4 regression test selection process

The inputs and output of the I-BACCI process are shown in gray blocks in Figure 3.1. The input artifacts to the process are the binary code of the COTS components (old and new versions); the source code and test suite of the development application; and all test cases which are mapped to the glue code functions they execute. These

input artifacts are generally available to users of COTS components. The output of the I-BACCI process is a reduced suite of regression test cases necessary to exercise the changed areas in the new COTS components.

The I-BACCI process has been evolved to Version 4 through the application of the process on both COFF and PE components written in C/C++. A summary of the evolution for the I-BACCI Versions 1 - 4 is shown in Table 3.1.

Table 3.1: Evolution of the I-BACCI process

Version Number	New features
1	Initial description based upon analysis of COFF components; the Decomposer and Trivial Information Zapper (D-TIZ) tool was created to perform the decomposition and remove trivial information [87].
2	Trivial Identifier of Differences in BInary-analysis Text Zapper (TID-BITZ) tool was created to reduce false positives due to shifted address and register changes [85].
3	Addition of the Call-graph Analyzer - Affected Function Identifier (CAAFI) tool to save analysis time and resources [86].
4	Generalize to support both COFF and PE components. Tools were integrated and improved to Pallino, a comprehensive automation. [88]

3.2 THE PROCESS STEPS

The first step of the I-BACCI process is to decompose the binary files of the component. The term *decomposing* is used here to refer to breaking up the binary code down into constituent elements, such as code sections and relocation tables. Prior to distribution, component source code is compiled into binary code, such as `.lib` and `.dll` files. Information on the data structure, functions, and function calling relationships of the source code is stored in the binary files according to

pre-defined formats, such as COFF and PE format, so that an external system is able to find and call the functions in the corresponding code sections. Often the first step can be accomplished by parsing tools available for the language/architecture. For example, COFF and PE binary files can be examined by the Microsoft COFF Binary File Dumper (DUMPBIN) [53]. Examples of DUMPBIN output will be shown in Chapter 6.

The second step, filtering trivial information, is frequently necessary because the output from the first step may contain trivial information such as timestamps and file pointers that are irrelevant to the change identification. Generally, the second step cannot be completed via existing tools. Therefore, the Pallino tool was created to perform the decomposition and remove trivial information. The output of the second step is the raw code section of each function/data, and function/data calling relationships for the new version of the component.

The main objective of **the third step** of the I-BACCI process is to identify true positive changes in the raw binary code of functions and data. The Pallino tool removes the false positives caused by differences due to trivial changes, such as shifted addresses and register reallocations. The algorithm used in Pallino will be introduced in detail in Chapter 4. Also, this step generates call graphs for the new version of the component. The call graphs can be drawn using graph generation tools such as GraphViz[3]. Pallino also represents and analyzes the call graphs of components of both COFF and PE types automatically in the I-BACCI Version 4. The call graph

[3] An open source tool, http://www.graphviz.org/

example of Pallino output will be shown in Figure 4.5, 4.6 and 4.7 in Chapter 4.1.

For different types of components, the execution order of the second step and the third step may be different. For example, for COFF components, the second step is executed before the third step. But the third step should be executed before the second step for PE components because only the names of exported component functions can be obtained in the binary code, such that functions have to be mapped between two releases after generating the call graphs for exported component functions. The main goal of the second and third steps is to facilitate comparisons and the identification of affected exported component functions.

In **the fourth step**, we identify changed and new added component functions according to the results of prior steps, and then identify affected exported component functions by tracing along the call graphs within the component using directed graph theory algorithms. Analysis starts from each component function identified as changed, and that change is propagated along the call graphs from the third step until the exported functions are reached. The output of the fourth step is a list of all affected exported component functions.

With the source code of glue code functions, **the fifth step** is to generate function call graphs for glue code functions that call exported component functions. The call graphs generated from the third step and the fifth step can be integrated together to learn how glue code functions are affected by changed and new component functions. Currently the call graph is similar to the example of Pallino output will be shown in

Figure 4.5, 4.6 and 4.7 in Chapter 4.1. The call graphs for the glue code can be drawn using existing open source tools such as Doxygen[4].

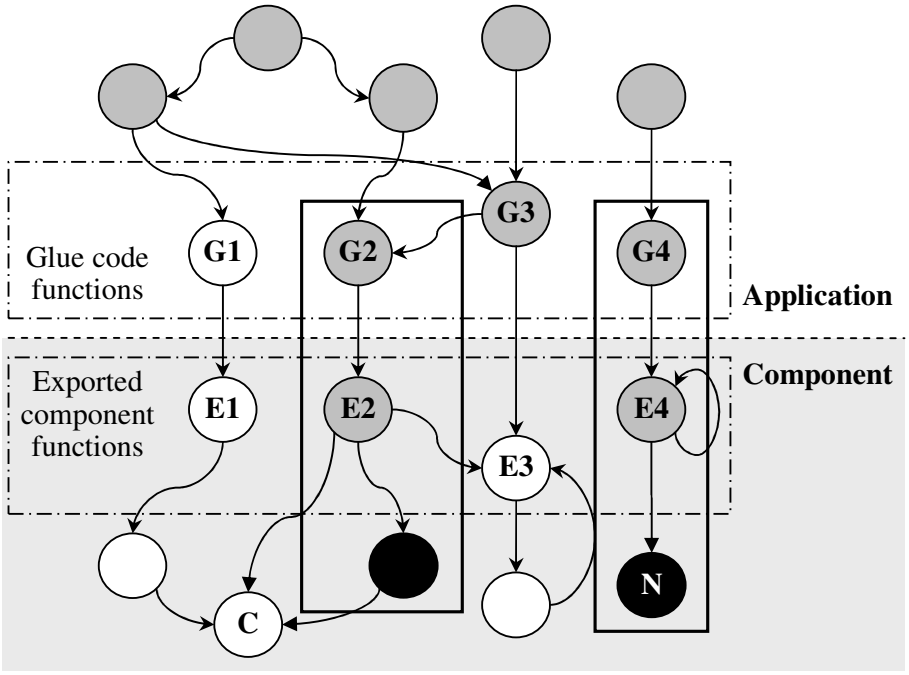
Similar to the fourth step, the affected glue code functions are identified in **the sixth step**. *Affected glue code functions* are functions within the glue code that directly call affected exported component functions. These are the functions within the application that are potentially affected by the changed function(s) in the component, and therefore need to be re-tested.

In **the seventh step**, the set of test cases which are mapped to the glue code functions they execute are used to select only test cases that cover the affected glue code functions, as identified by the steps above.

3.3 ILLUSTRATION OF THE PROCESS

The I-BACCI process has the potential to reduce the set of regression test cases because it focuses on the affected glue code functions and ignores the unaffected areas in the application. The process is illustrated in Figure 3.2. Changed and new added component functions are identified, as shown in the black circles. Then, the analysis starts from each component function identified as "changed" or "new," backtracks the call graphs to identify all functions that directly or indirectly call the changed functions, until the glue code functions that can be reached within one edge from exported component functions in the call graph are reached.

[4] An open source tool, http://www.stack.nl/~dimitri/doxygen/



Legend

- Each circle stands for a *function*. Each arrow connector stands for a *function call*.
- The dotted line comparts *application* and *component* functions.
- Black circles stand for the *changed functions*.
- Black circles with letter 'N' stand for the *added functions*.
- Gray circles stand for the *affected functions* (unchanged functions that are in the call chain of the changed or added component functions).
- White circles stand for the *unchanged and unaffected functions*.
- Circles with letter 'G' and 'E' stand for the *glue code functions* and *exported component functions*, respectively.
- Real line squares stand for the *firewalls*. Glue code functions in the firewall are the *affected glue code functions*.

Figure 3.2: Illustration of the use of firewall RTS in the I-BACCI process

In this example, exported component functions E1 and E3 are not in the call chain of either changed or new component functions, such that are not affected and are shown as white circles. However, exported component functions E2 and E4 are affected by changed and new component functions which are shown as black circles.

Therefore, exported component functions E2 and E4 are considered as “changed”, and glue code functions in the solid line box (G2 and G4) that call E2 and E4 are the affected glue code functions, which need to be re-tested. Although also affected by the component change, glue code function G3 is not chosen to be re-tested because G2 has been re-tested, according to the firewall analysis concept.

Compared to Figure 2.2 in Chapter 2.3, the I-BACCI process extends the traditional concept and scope of application for firewall analysis for use with COTS components. In my approach, firewall analysis is applied on the application instead of within the component. If the traditional firewall is used within the component, a significant number of exported component functions might be considered as "affected" by the change impact analysis. In this example, the traditional firewall would consist of functions E2, E4, and C, and the changed and new added component functions. The user of the component does not have any source code and test cases for these functions, and therefore can not conduct user-oriented component testing. Analysis needs to backtrack the execution traces to find out which exported component functions are in the call chain of the component functions that are within the firewall. In this example, all exported component functions (E1, E2, E3, and E4) would be considered as "affected". My approach extracts the execution traces from glue code to changed or new functions in binary code, i.e., only considers the component functions that directly or indirectly call changed or new functions. The reason that my approach ignores the component functions that are called by the

changed or new functions in binary code is that, only those tests that execute the changed part (instead of all affected part) of the component need to be selected. Traditional firewall analysis is then applied on source code of the application that incorporates the component, i.e., test cases that execute the glue code of the application are selected. In this example, two exported component functions (E2 and E4) would be considered as "affected" using my approach.

3.4 LIMITATIONS

The I-BACCI process shares an acknowledged technical limitation with all existing firewall methods: the potential for reporting false positives and false negatives in situations where binary differences are due to factors other than changes in source code (e.g. build tools, environment, or target platform). Although the I-BACCI process does work with the binary files for the component, the current method of analysis precludes identification of such differences.

The second limitation of the I-BACCI process is its potential for identifying false positives by assuming, in tracing the call graphs, that any uses of called functions with changed binaries will be affected by the change. However, an actual use of a changed function might never exercise the changed logic or data. With further development of the I-BACCI process, these unneeded tests may be eliminated from the regression suite.

Also, as will be noted in Chapter 6, license agreement considerations may constrain the breadth of applicability of this tool and method.

Finally, the I-BACCI process requires as input the test suites with traceability to the glue code functions they execute, in order to perform RTS.

CHAPTER 4

SUPPORTING AUTOMATION: PALLINO

This chapter presents the comprehensive supporting automation for the I-BACCI process, Pallino. Pallino performs static binary change identification and impact analysis and can be modified to support other RTS methods for COTS components. The input artifacts to Pallino are the binary code of the components (old and new versions), which is generally available to users of COTS components. Pallino outputs a list of affected exported component functions.

Based on the list of affected exported component functions and the original test suite, testers can identify glue code functions that call affected exported component functions. Then the subset of the regression test cases that execute the glue code which is affected by the changed areas in the new COTS components can be identified. Currently Pallino works on components in Common Object File Format (COFF) or Portable Executable (PE) formats written in C/C++.

The remainder of this chapter is structured as follows. Chapter 4.1 illustrates two examples of how Pallino works. Chapters 4.2 and 4.3 describe the architecture and algorithms of Pallino, respectively. Chapter 4.4 presents how to use Pallino.

The limitations of Pallino are discussed in Chapter 4.5.

4.1 ILLUSTRATION OF USE

This chapter presents two examples to illustrate how affected exported component functions are identified from binary code. The examples for COFF component and PE component will be discussed in Chapter 4.1.1 and Chapter 4.1.2, respectively.

4.1.1 ILLUSTRATION OF USE WITH COFF COMPONENT

Windows NT® uses a special format for the executable (image) files and object files. The format used in these files is referred to as COFF files[5]. Object files created from C or C++ programs using many compilers conform to COFF, including the Visual C++ and the GNU Compiler Collection (GCC).

In the example to follow, Release 4 and Release 5 of an ABB component are compared. These two releases are referred to as the "*old*" and the "*new*" releases. The input to Pallino is `.lib` binary files of the two releases. Binary code fragments in the two releases are shown in Figure 4.1.

At first glance, we can not see any relationship between the two binary code fragments because both the binary code and the address ranges are different. The DUMPBIN tool was used to translate the binary files into plain text. The counterparts in the output of DUMPBIN for the two binary code fragments are shown in Figure 4.2.

[5] MSDN Library - Visual Studio .NET 2003

```
00003830: ......                                  /* Old Release */
00003840: 00 07 00 51 56 8B 74 24 0C 57 C6 44 24 0B 01 83
00003850: 7E 04 01 7D 07 5F 83 C8 FF 5E 59 C3 56 E8 00 00
00003860: 00 00 8B F8 83 C4 04 85 FF 74 15 6A FF 68 00 00
00003870: 00 00 E8 00 00 00 00 83 C4 08 8B C7 5F 5E 59 C3
00003880: 56 E8 00 00 00 00 8B F8 83 C4 04 85 FF 74 18 68
00003890: 80 00 00 00 68 00 00 00 00 E8 00 00 00 00 83 C4
000038a0: 08 8B C7 5F 5E 59 C3 56 E8 00 00 00 00 8B F8 83
000038b0: C4 04 85 FF 74 15 6A FF 68 00 00 00 00 E8 00 00
000038c0: 00 00 83 C4 08 8B C7 5F 5E 59 C3 8B 4E 04 8D 44
000038d0: 24 0B 6A 01 50 6A 04 68 FF FF 00 00 51 FF 15 00
000038e0: 00 00 00 85 C0 74 1B 6A 04 68 00 00 00 00 E8 00
000038f0: 00 00 00 6A 04 68 00 00 00 00 E8 00 00 00 00 83
00003900: C4 10 56 E8 00 00 00 00 83 C4 04 5F 5E 59 C3 90
00003910: 90 90 90 1B 00 00 00 96 00 00 00 14 00 2B 00 00
00003920: ......
00003a60: ......                                  /* New Release */
00003a70: 00 01 00 00 14 00 00 00 00 A5 00 00 00 07 00 51
00003a80: 56 8B 74 24 0C 57 C6 44 24 0B 01 83 7E 04 01 7D
00003a90: 07 5F 83 C8 FF 5E 59 C3 56 E8 00 00 00 00 8B F8
00003aa0: 83 C4 04 85 FF 74 15 6A FF 68 00 00 00 00 E8 00
00003ab0: 00 00 00 83 C4 08 8B C7 5F 5E 59 C3 56 E8 00 00
00003ac0: 00 00 8B F8 83 C4 04 85 FF 74 18 68 80 00 00 00
00003ad0: 68 00 00 00 00 E8 00 00 00 00 83 C4 08 8B C7 5F
00003ae0: 5E 59 C3 8B 4E 04 8D 44 24 0B 6A 01 50 6A 04 68
00003af0: FF FF 00 00 51 FF 15 00 00 00 00 85 C0 74 1B 6A
00003b00: 04 68 00 00 00 00 E8 00 00 00 00 6A 04 68 00 00
00003b10: 00 00 E8 00 00 00 00 83 C4 10 56 E8 00 00 00 00
00003b20: 83 C4 04 5F 5E 59 C3 90 90 90 90 90 90 90 90 1B
00003b30: ......
```

Figure 4.1: Binary code fragments in the two releases of a COFF component

The sections can be located and identified by function signatures, e.g., `function1` in this example. Directive information, such as size of raw data and function signature, is shown in the "`SECTION HEADER`" subsection. The "`RAW DATA`" subsection displays the binary code that represents `function1` for each release, i.e. the code in boldface in Figure 4.1. The "`RELOCATIONS`" subsection lists

which other functions and data are called by `function1`. Non-trivial difference for `function1` between the two releases is underscored in Figure 4.1 and 4.2. Thirty six bytes of code (three lines of source code), with three functions and data calls were deleted in the new release.

```
SECTION HEADER #31                          /* Old Release */
   .text name                          // section name
        ......                         // directive information
        Communal; sym= _function1           // signature
        ......                         // directive information
RAW DATA #31
00000000: 51 56 8B 74 24 0C 57 C6 44 24 0B 01 83 7E 04 01
00000010: ......          // more binary code for function1
RELOCATIONS #31
Offset    Type  Symbol Index Symbol Name
--------  ----- ------------ -----------
0000001B REL32            96 _function2
0000002B DIR32            B8 string_data1
00000030 REL32           152 _function3
0000003F REL32            9D _function4
00000052 DIR32            B5 string_data2
00000057 REL32           152 _function3
00000066 REL32            A4 _function5
00000076 DIR32            B2 string_data3
0000007B REL32           152 _function3
0000009C DIR32            6C _function6
000000A7 DIR32            AF string_data4
000000AC REL32           152 _function3
000000B3 DIR32            AC string_data5
000000B8 REL32           152 _function3
000000C1 REL32            BD _function7
```

```
SECTION HEADER #31                          /* New Release */
   .text name                          // section name
        ......                         // directive information
        Communal; sym= _function1           // signature
        ......                         // directive information
RAW DATA #31
00000000: 51 56 8B 74 24 0C 57 C6 44 24 0B 01 83 7E 04 01
00000010: ......          // more binary code for function1
RELOCATIONS #31
```

```
Offset    Type  Symbol Index Symbol Name
--------  ----- ------------ -----------
0000001B REL32            97 _function2
0000002B DIR32            B6 string_data1
00000030 REL32           14F _function3
0000003F REL32            9E _function4
00000052 DIR32            B3 string_data2
00000057 REL32           14F _function3
00000078 DIR32             C _function6
00000083 DIR32            B0 string_data4
00000088 REL32           14F _function3
0000008F DIR32            AD string_data5
00000094 REL32           14F _function3
0000009D REL32            BB _function7
```

Figure 4.2: DUMPBIN output of the two releases of a COFF component

Using the calling relationship information in the "RELOCATIONS" subsection, Pallino generates call graphs and identifies the affected exported component functions in the new release. Figure 4.3 shows how function1 (in black) affects glue code.

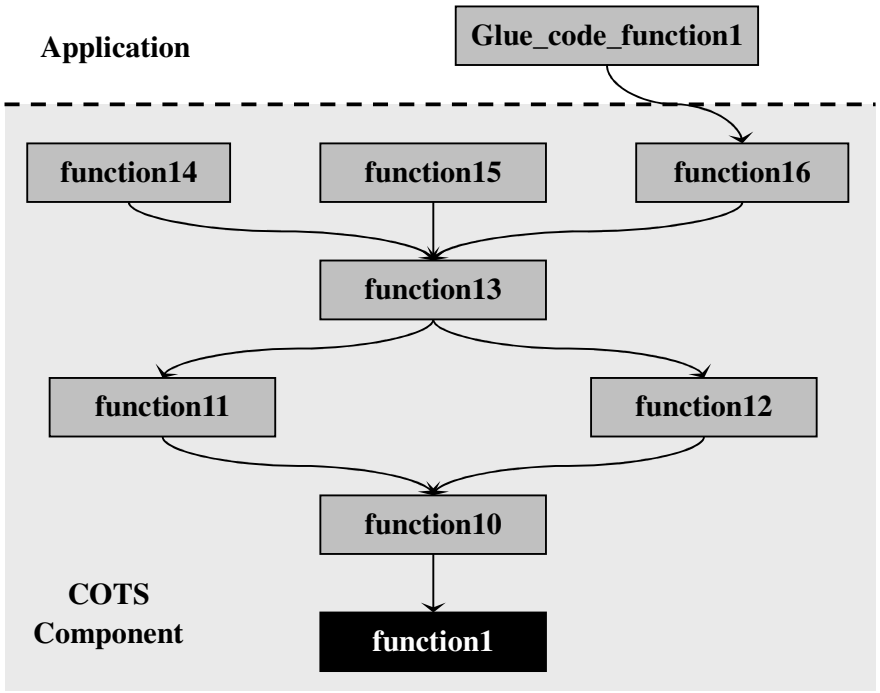


Figure 4.3: Call graph: how component change affects glue code

Although three exported component functions are affected by `function1`, only one glue code function (`Glue_code_function1`) calls one of the affected exported component functions. Therefore, RTS will only need to select test cases for `Glue_code_function1` from the initial test suite.

4.1.2 ILLUSTRATION OF USE WITH PE COMPONENT

Many executables, such as `.exe` files, Object Linking and Embedding Control Extension (OCX) controls, and Control Panel applets (`.cpl` files) are in PE format. When loaded into main memory by the Windows loader, PE files can be mapped directly into memory, such that the data structures on disk are the same as those Windows uses at runtime. If one knows how to find something in a PE file, he can almost certainly find the same information when the file is loaded in memory. [59] This attribute of PE files facilitates static BCA.

However, some characteristics of the PE format make the change identification and call graph generation more complex than analyzing COFF files. For example, only the names of exported component functions can be obtained in the binary code, such that functions have to be mapped between two releases after generating the call graphs for exported component functions. Also, the DUMPBIN output for PE files contains only one *.text* section where the raw code fragments for all the functions are consecutively arranged. Relocation information is stored in the only *.reloc* section, as shown in Figure 4.4. Whereas the information for each function, such as raw code fragment and relocation table, locates in separated sections in the DUMPBIN output

for COFF files. Pallino needs to parse these sections and integrate information to achieve the goal of change identification and impact analysis.

This example illustrates how the exported function `foo` is affected by changed data. Information related to `foo` in the DUMPBIN output for the two `.dll` files are shown in Figure 4.4. The *Relative Virtual Address (RVA)* of the raw code segment of `foo` can be found in the exports table in the *.rdata* section, e.g., `00003BC0` in the old release and `000024D0` in the new release. Therefore, the start virtual addresses of the raw code segment of `foo` can be calculated by adding the image base address (`0x10000000` in this example) to the RVA, i.e., `0x10003BC0` in the old release. The binary code that represents `foo` for each release is in boldface in the "`RAW DATA #1`" subsections. Differences for the raw code segment of `foo` between the two releases are gray highlighted in Figure 4.4.

```
SECTION HEADER #1                              /* Old Release */
   .text name                          // code section
         ......                        // directive information
RAW DATA #1
......
10003BC0: 8B 44 24 04 85 C0 74 13 8B 4C 24 08 51 68 34 82
10003BD0: 01 10 50 E8 88 6D 00 00 83 C4 0C 8B 44 24 0C 85
10003BE0: C0 74 13 8B 54 24 10 52 68 2C 82 01 10 50 E8 6D
10003BF0: 6D 00 00 83 C4 0C 8B 44 24 14 85 C0 74 13 8B 4C
10003C00: 24 18 51 68 24 82 01 10 50 E8 52 6D 00 00 83 C4
10003C10: 0C B8 01 00 00 00 C2 18 00 90 90 90 90 90 90 90
......
SECTION HEADER #2
  .rdata name                        // read only data section
         ......                      // directive information
  ordinal hint RVA      name         // exports table
        ......
        6    A 00003BC0 foo
        ......
```

```
SECTION HEADER #3
   .data name                    // read/write data section
         ......                  // directive information
RAW DATA #3
......
10018220: 00 00 00 00 41 53 43 49 49 00 00 00 31 2E 31 2E
10018230: 38 00 00 00 45 42 50 41 5F 4C 49 43 45 4E 53 49
......
SECTION HEADER #4
  .reloc name                    // relocation section
         ......                  // directive information
BASE RELOCATIONS #4
     ......
    3000 RVA
     BCE  HIGHLOW     10018234             // call data3
     BE9  HIGHLOW     1001822C             // call data2
     C04  HIGHLOW     10018224             // call data1
     ......
```

```
SECTION HEADER #1                          /* New Release */
   .text name                    // code section
         ......                  // directive information
RAW DATA #1
......
100024D0: 8B 44 24 04 85 C0 74 13 8B 4C 24 08 51 68 08 7B
100024E0: 01 10 50 E8 78 84 00 00 83 C4 0C 8B 44 24 0C 85
100024F0: C0 74 13 8B 54 24 10 52 68 00 7B 01 10 50 E8 5D
10002500: 84 00 00 83 C4 0C 8B 44 24 14 85 C0 74 13 8B 4C
10002510: 24 18 51 68 F8 7A 01 10 50 E8 42 84 00 00 83 C4
10002520: 0C B8 01 00 00 00 C2 18 00 90 90 90 90 90 90 90
......
SECTION HEADER #2
  .rdata name                    // read only data section
         ......                  // directive information
  ordinal hint RVA      name     // exports table
        ......
        6    A 000024D0 foo
        ......
SECTION HEADER #3
   .data name                    // read/write data section
         ......                  // directive information
RAW DATA #3
......
10017AF0: 25 30 38 6C 78 00 00 00 41 53 43 49 49 00 00 00
```

```
10017B00: 31 2E 31 2E 39 00 00 00 45 42 50 41 5F 4C 49 43
......
SECTION HEADER #4
  .reloc name                    // relocation section
        ......                   // directive information
BASE RELOCATIONS #4
     ......
    2000 RVA
     4DE  HIGHLOW     10017B08              // call data3
     4F9  HIGHLOW     10017B00              // call data2
     514  HIGHLOW     10017AF8              // call data1
     ......
```

Figure 4.4: DUMPBIN output of the two releases of a PE component

These differences are all trivial shifted addresses to be ignored in semantic differencing. However, according to the information in the relocation sections, `foo` calls `data2`, which changes from `0x312E312E38` (ASCII string "`1.1.8`") to `0x312E312E39` (ASCII string "`1.1.9`"), as shown in underscored code in the "`RAW DATA #3`" subsections. The one byte change exactly reflected the modification in source code: the value of macro definition "`VERSION`" changed from "`1.1.8`" to "`1.1.9`" in the new release.

Pallino then generates call graphs and identifies how changed `data2` affects `foo`. The generated call graph is initially recursively represented in hierarchical plain text, as shown in Figure 4.5.

```
foo
   --> SUB_10017B08   //data1 called by foo
   --> SUB_1000A960   //noNameFunction1 called by foo
   --> SUB_10017B00   //data2 called by foo
   --> <SUB_1000A960> //noNameFunction1 called by foo again
   --> SUB_10017AF8   //data3 called by foo
   --> <SUB_1000A960> //noNameFunction1 called by foo again
```

Figure 4.5: Generated call graph initially represented in plain text

Pallino also generates another complex hierarchical representation of the call graph for `foo`, including all raw code of sub-functions and/or data that might be called by that exported function, as shown in Figure 4.6.

```
 [foo] 8B 44 24 04 85 C0 74 13 8B 4C 24 08 51 68
   --> [SUB_10017B08] 45 42 50 41 5F 4C 49 43 45 4E 53 49 4E
47 20 4C 69 62 72 61 72 79 00 //data1
[foo] 50 E8
   --> [SUB_1000A960] 8B 4C 24 0C 57 85 C9 74 7A 56 53 8B D9
8B 74 24 14 F7 C6 03 00 00 00 8B 7C 24 10 75 07 C1 E9 02 75
6F EB 21 8A 06 46 88 07 47 49 74 25 84 C0 74 29 F7 C6 03 00
00 00 75 EB 8B D9 C1 E9 02 75 51 83 E3 03 74 0D 8A 06 46 88
07 47 84 C0 74 2F 4B 75 F3 8B 44 24 10 5B 5E 5F C3 F7 C7 03
00 00 00 74 12 88 07 47 49 0F 84 8A 00 00 00 F7 C7 03 00 00
00 75 EE 8B D9 C1 E9 02 75 6C 88 07 47 4B 75 FA 5B 5E 8B 44
24 08 5F C3 89 17 83 C7 04 49 74 AF BA FF FE FE 7E 8B 06 03
D0 83 F0 FF 33 C2 8B 16 83 C6 04 A9 00 01 01 81 74 DE 84 D2
74 2C 84 F6 74 1E F7 C2 00 00 FF 00 74 0C F7 C2 00 00 00 FF
75 C6 89 17 EB 18 81 E2 FF FF 00 00 89 17 EB 0E 81 E2 FF 00
00 00 89 17 EB 04 33 D2 89 17 83 C7 04 33 C0 49 74 0A 33 C0
89 07 83 C7 04 49 75 F8 83 E3 03 75 85 8B 44 24 10 5B 5E 5F
C3 //noNameFunction1
[foo] 83 C4 0C 8B 44 24 0C 85 C0 74 13 8B 54 24 10 52 68
   --> [SUB_10017B00] 31 2E 31 2E 39 00 //data2
[foo] 50 E8
   --> <SUB_1000A960> //noNameFunction1 (again)
[foo] 83 C4 0C 8B 44 24 14 85 C0 74 13 8B 4C 24 18 51 68
   --> [SUB_10017AF8] 41 53 43 49 49 00 //data3
[foo] 50 E8
   --> <SUB_1000A960> //noNameFunction1 (again)
[foo] 83 C4 0C B8 01 00 00 00 C2 18 00
```

Figure 4.6: Generated call graph in complex representation

All code is grey highlighted in Figure 4.6. The symbol "-->" indicates the calling direction. For example, foo calls data1, noNameFunction1, data2, and data3 in Figure 4.5 and 4.6. The signature of a non-exported component function or data, which can not be retrieved from the binary file, is represented by a string "SUB_" followed by

the start virtual address of this function or data. When has been represented, a function or data is then represented by its signature enclosed by a pair of brackets, e.g., <SUB_1000A960>. Figure 4.7 shows the counterpart graphical call graph. Glue code functions that call foo are then selected for re-testing.

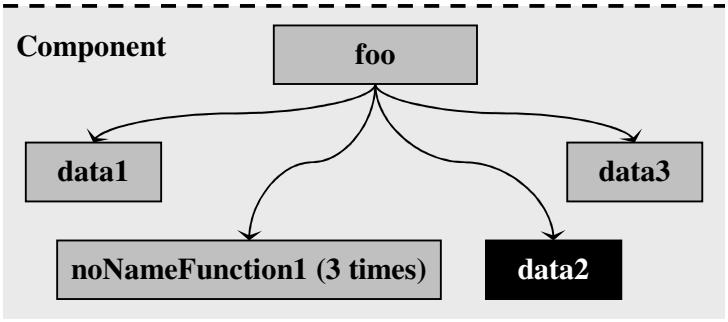


Figure 4.7: Call graph: How changed data affects exported component function

4.2 ARCHITECTURE

The overall architecture of Pallino conforms to the model-view-controller (MVC) model, as shown in Figure 4.8.

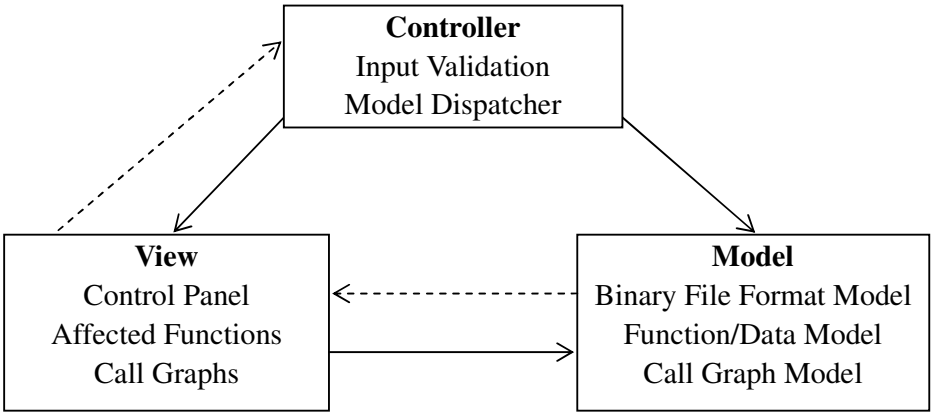


Figure 4.8: Overall architecture

The solid lines represent direct associations, and the dashed lines represent indirect associations. An MVC architecture separates data (model) and user

interface (view) concerns, so that changes to the user interface do not affect the data handling, and that the data can be reorganized without changing the user interface.

According to the object-oriented software design principle of "program to an interface, not an implementation" [23], a generic interface `BinaryFileFormatModel` was created to represent the abstract concept of the binary file format model. The concrete types of binary file format model, including `COFFFormat` and `PEFormat`, implement the generic interface and represent the data structure of COFF and PE binary file formats, respectively. The client code accesses objects of a concrete binary file format model only through their abstract interface. This pattern allows for new derived types of binary file format model to be introduced with no change to the code that uses the base object, increasing the extensibility of the tool. Functionality that processes other binary file format, such as Executable and Linking Format (ELF)[6], can be easily added into the system without affecting many other modules, as shown in Figure 4.9.

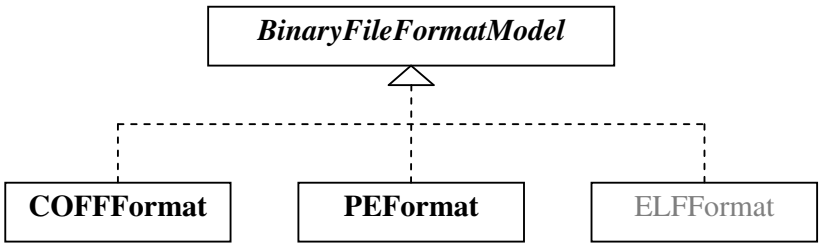


Figure 4.9: Binary File Format Model

Additionally, we adapted the data structures to the functionality of change identification and impact analysis. Not all information in the binary file formats are

[6] A common standard file format for executables, object code, shared libraries, and core dumps, which was chosen as the standard binary file format for Unix and Unix-like systems.

needed to be described in the models used in Pallino, for example, time date stamps and number of symbols. The data structure of PEFormat is shown in Figure 4.10.

```
Class PEFormat {
  PEFileHeader     fileHeader;    //File header
  PEOptionalHeader optHeader;     //Optional header
  PESectionHeader  textHeader;    //.text section header
  PESectionHeader  rdataHeader;   //.rdata section header
  String           rdataRaw;     //.rdata section data
  PESectionHeader  dataHeader;    //.data section header
  String           dataRaw;      //.data section data
  PESectionHeader  idataHeader;   //.idata section header
  String           idataRaw;     //.idata section dara
  PESectionHeader  relocHeader;   //.reloc section header
  PEExportTable    exportsTable;  //Exports table
  PEImportTable    importsTable;  //Imports table
}
```

Figure 4.10: PE format data structure

In the function/data model, functions and data are abstracted as the same class (`PEFunctionData`) with the following main attributes: signature, start virtual addresses, end virtual address, raw binary code, and relocation list. Functions without explicit signatures (e.g. non-exported functions in PE files) and all data use start virtual addresses as their signatures.

The view module of the architecture includes the control panel, and result representation and displaying, which will be described in Chapter 4.4 with the illustration of use.

The controller module is responsible for processing and responding the input event from the user interface. First, the controller validates the input and recognizes the type of the input binary file by the magic number to decide to which algorithm it

will pass the request. A *magic number* is a pre-defined constant, typically located at the first few bytes of a binary file, used to identify the file type. For example, PE files start with the ASCII string 'MZ' (0x4D5A), and the magic number of a COFF file is the ASCII string "!<arch>\n" (0x213C617263683E0A). The controller then executes the algorithm and accesses the corresponding model. Finally, the results are produced and returned to the view module and user interface.

4.3 ALGORITHMS

In this chapter, the algorithms that were developed for components in COFF and PE formats are described, respectively.

4.3.1 ALGORITHMS FOR COFF COMPONENTS

At first, DUMPBIN was invoked to convert the binary library code into plain text. An example of the DUMPBIN output is shown in the Figure 4.11.

```
int ClassA::functionA(int s) {          /* Source Code */
   return state==s;
}
```

```
SECTION HEADER #78/* Corresponding section in Old Release */
   .text name
       0 physical address
       0 virtual address
      20 size of raw data
    722B file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60501020 flags
         Code
         Communal; sym= "public: virtual int __thiscall
ClassA::functionA(int)"
         16 byte align
         Execute Read
```

```
RAW DATA #78
00000000: 8B 89 A0 06 00 00 8B 54 24 04 33 C0 3B CA 0F 94
00000010: C0 C2 04 00 90 90 90 90 90 90 90 90 90 90 90 90
SECTION HEADER #79/* Corresponding section in New Release */
   .text name
       0 physical address
       0 virtual address
      20 size of raw data
    73D1 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60501020 flags
         Code
         Communal; sym= "public: virtual int __thiscall
ClassA::FunctionA(int)"
         16 byte align
         Execute Read
RAW DATA #79
00000000: 8B 89 CC 06 00 00 8B 54 24 04 33 C0 3B CA 0F 94
00000010: C0 C2 04 00 90 90 90 90 90 90 90 90 90 90 90 90
```

Figure 4.11: Source code and DUMPBIN output for functionA in ClassA

Function names, binary code representation of the functions, and relocation tables are all clearly described in the output text of DUMPBIN. The algorithm scans the output of DUMPBIN, saves the code sections of functions into separate files, and collects and saves the relocation tables of the functions into a text file (henceforth called "*relocation table set*"). The function list is fed into next step to perform differencing. The relocation table set is utilized to generate and analyze call graphs of the components in later steps.

In the next step, it is necessary to reduce the number of false positive changes identified due to trivial changes, such as shifted addresses and register reallocations. A

large number of false positives were observed in the initial case study of the I-BACCI Version 1 [87], which increased the number of glue code functions that were identified for retesting. To explore the cause of the false positives, the analyzer examined the source code and the associated binary library files of the component. A large amount of false positives were caused by changes in registers used and addresses of variables and functions, which typically would not cause functional changes in the code.

For example, as shown in bold in the Figure 4.11, the binary code `8B89A0060000` means "copy the operand in the address of register ECX plus offset `0x06A0` to register ECX", where `8B89` is the opcode of the instruction and `A0060000` is the address offset. Therefore, in this example, the only difference in binary is that the address offset was changed from `A0060000` to `CC060000`. Further examination of the source code showed that seven new function declarations and one new variable definition were added before the variable state was defined in one of the header files included in the source file of the new release. As a result, the offset of the variable state was changed accordingly. In this case, the binary code change identified is not a real change and can be ignored in the change identification. The binary code like `8B89A0060000` is called an example of a "*binary code comparison false positive pattern.*"

Many such false positive patterns were found in the first case study. The full list of these empirical patterns is shown in Appendix A. False positive patterns are

identified by their prefix. The prefix of a pattern can be the opcode of an instruction (e.g., `FF50`), or first few bits of an opcode (e.g., `8B8`) which means all opcode that begins with these bits are prefixes of false positive patterns (e.g., from `8B80` to `8B8F`). The algorithm scans the two versions of raw binary code of a function. For each false positive pattern, when the prefix of the pattern is found, the corresponding numbers of bytes from the start byte of the prefix are marked as a constant symbol (e.g., "_") in the raw code. Only if the remaining bytes of the two versions of binary code are the same, the function is considered as unchanged. The algorithm reduced the false positive rate to less than 8% in the case studies [86].

However, this algorithm may introduce false negatives by eliminating real code changes [85]. As shown in Table 4.1, there were no false negative without using the algorithm. The use of the algorithm introduced less than 2% false negatives in our case studies. The high false positive rate without the algorithm would lead to much more affected functions in both component and application.

The algorithm then builds and analyzes the call graphs of components of COFF type automatically using the relocation table set generated in the first step, and changed functions identified in the second step. The relocation table set of a component is converted into an adjacency-matrix [19] to represent call graphs of the functions in the component. For each changed function, the algorithm then backtracks the call graphs to identify all functions that directly or indirectly call the changed function.

Table 4.1: Analysis of false positives and false negatives

(FP is false positive; FN is false negative.)

Comparisons	% of FPs without the algorithm	% of FPs with the algorithm	% of FNs without the algorithm	% of FNs with the algorithm
A1 vs. A2	90.1%	5.6%	0%	0%
A2 vs. A3	0%	0%	0%	0%
A3 vs. A4	0%	0%	0%	0%
A4 vs. A5	0%	0%	0%	0%
A5 vs. A6	0%	0%	0%	0%
B1 vs. B2	59.3%	4.9%	0%	0.5%
B2 vs. B3	12.4%	6.1%	0%	1.6%
B3 vs. B4	0%	0%	0%	0%
B4 vs. B5	76.9%	7.7%	0%	0%

The outputs of Pallino for analyzing COFF components include: (1) the call graph of each exported component function; (2) a differencing report on the two releases; and (3) a list of all affected exported component functions in the new release.

4.3.2 ALGORITHMS FOR PE COMPONENTS

The algorithm examines the binaries from coarse to fine granularity step by step. First, the tool invokes DUMPBIN to translate the illegible binary library files into readable plain text files. This file-level granularity step assumes that file names do not change between releases. Then a file reader automatically scans the DUMPBIN output and loads useful information, such as instructive information in file header, section headers, exports table and imports table, into the predefined data structure `PEFormat` which is constructed according to the PE file format specification. File and section information is ready to facilitate future lookup after this section-level step.

The next finer granularity is in function/data-level. Binary code of functions and

data are stored consecutively in *.text* section and data sections (*.rdata*, *.data*, *.idata*, etc.), respectively. However, only names of exported component functions are available. Other functions and all data have to be labeled by their start virtual addresses. The data structure `PEFunctionData`, as discussed in Chapter 4.2, is used to represent the functions and data. The relocation table is read from the *.reloc* section and then converted into a `Hashtable` called *relocation index*. For each key-value pair in the relocation index, the key is a *calling virtual address* where the control flow jumps to another function or data, and the start virtual address of the function or data being called (a.k.a. *target virtual address*) is stored as the value. Function calling virtual address and target virtual address can also be calculated according to the position of each call instruction and the address offset following each call instruction, respectively. Because only binary code is available instead of assembly code, the tool searches opcode `E8` and `E9` which represent "call near" in the Intel instruction set[7] to locate the position of each function call. After finding all functions and data start virtual addresses, an array in `PEFunctionData` type is constructed and the raw code of the *.text* and data sections is decomposed into separate functions or data.

The function/data call graphs and full code representation for all exported component functions can be generated recursively following the calling track. A few steps that remove trivial bytes are also conducted during processing of this level. For

[7] Intel® Architectures Software Developer's Manuals, http://developer.intel.com/products/processor/manuals/index.htm

example, most raw code of functions/data is followed by a few useless bytes (e.g. `90`, `CC`) for the purpose of alignment.

Further instruction-level comparisons can be conducted after the function/data level if the full code representations of an exported component function in two releases are still different. For example, false positives may be caused by register allocation changes from build to build. After all of the above steps, a report on differencing of exported component functions is generated. This report can be used to identify affected application code and then select proper regression test cases.

The outputs of Pallino for analyzing PE components include: (1) the call graph of each exported component function; (2) a full binary code representation of each exported component function, including all code of sub-functions and data that might be called by that exported function; (3) a differencing report on the two releases; and (4) a list of all affected exported component functions in the new release.

4.4 USING PALLINO

Although developed in Java, Pallino is transformed to a Windows executable file (`.exe`) by exe4j[8] to facilitate the use in the Windows operating environment. An illustrative screen shot of the main console of Pallino is shown in Figure 4.12.

There are three panes on the main console: input pane, run pane, and results pane. The user of Pallino first specifies the binary files of both old and new versions of a component, and a working directory in the input pane. The specified working directory is for the purpose of saving results and running log. Once the input is

[8] http://www.ej-technologies.com/products/exe4j/overview.html

specified and the "Start to Run" button in the run pane is clicked, Pallino accepts and validates the input, executes the corresponding algorithms according to the file format, and upon completion, refreshes the results in the results pane.

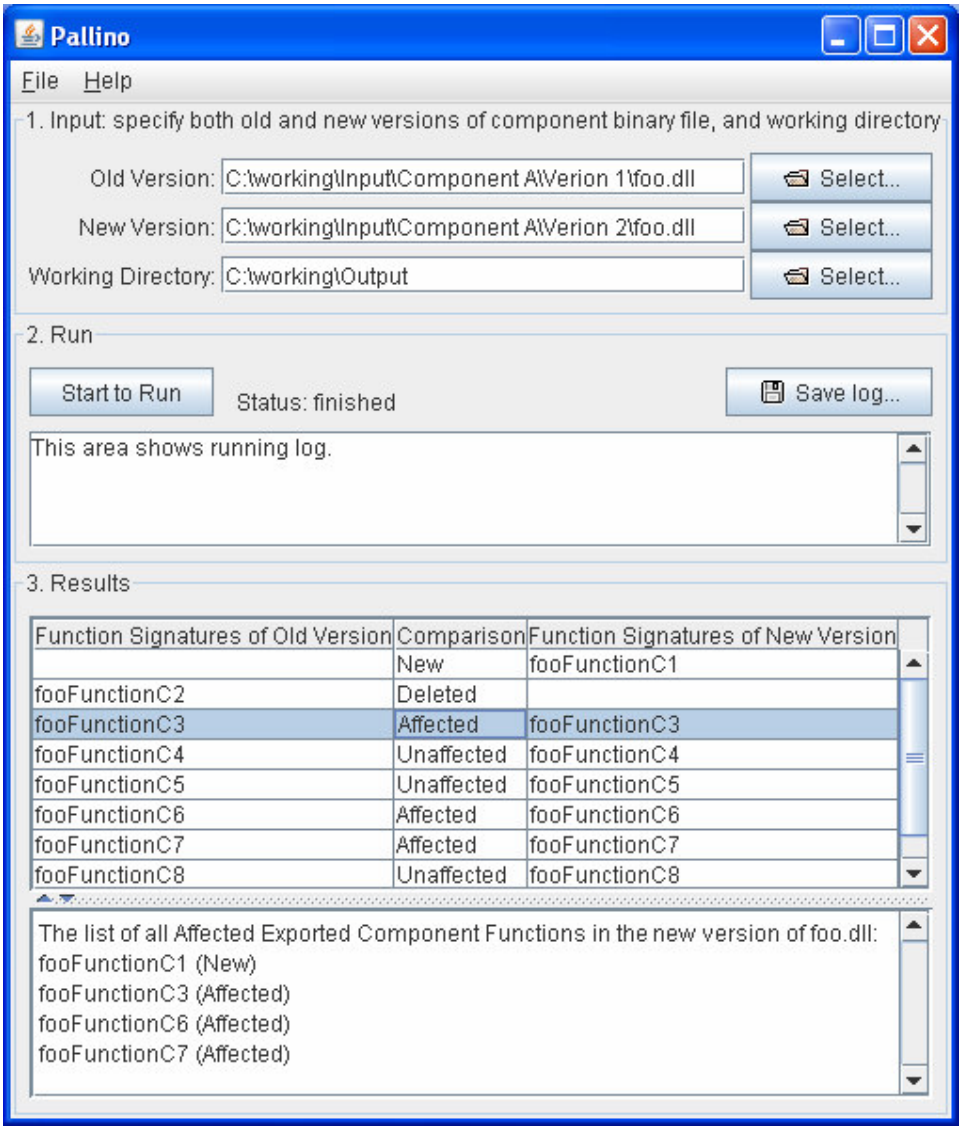


Figure 4.12: Pallino screen shot

The exported component functions for both versions of the component are shown in a table in the results pane, matched by the function signatures, i.e., functions with the same signatures are shown in the same row. The user can clearly see which

functions are new added, changed, affected, or unaffected in the new version of the component in the middle column of the result table. Further explanation, including a list of all affected exported component functions for the new version of the component, is shown in a text area in the results pane. The results are also saved into files for the RTS analysis of the I-BACCI process. The running log can also be saved to a file by clicking on the "Save log..." button.

4.5 LIMITATIONS

Pallino works only when the releases of components are built by the same compiler. If two compared releases are built by different compilers or linkers, Pallino will yield a significant number of false positives.

CHAPTER 5

CASE STUDIES

The I-BACCI Version 4 process was applied to four industrial case studies to identify reduced test suites. A fifth case study was conducted to evaluate the total time costs for regression testing, i.e., the overall efficiency, of firewall analysis RTS technique which is used in the I-BACCI process by comparing with the modified entity RTS techniques.

The first case study (henceforth called Case 1) was conducted on a 757 thousand lines of code (KLOC) ABB application (henceforth called Application A) written in C/C++. Application A uses a 67 KLOC internal ABB software component (henceforth called Component A) of .lib files written in C. Six incremental releases of Component A were analyzed and compared (henceforth referred to as Release A1 through Release A6, respectively). Each Component A release is a library file with size of about 800 kilobyte.

A second case study (henceforth called Case 2) was conducted on a 40 KLOC ABB application (henceforth called Application B) written in C/C++. This product uses a 300 KLOC internal ABB software component (henceforth called Component B) of .lib files written in C. Five incremental releases of Component B were analyzed

and compared (henceforth referred to as Release B1 through Release B5, respectively). Each Component B release contains eight libraries with total size of 1.39 ~ 1.65 megabyte. The full, retest-all strategy takes over four man months of effort to run.

The third case study (henceforth called Case 3) was conducted with the same application in Case 1 (Application A). But for Case 3, Application A uses another three KLOC internal ABB software component (henceforth called Component C) of a DLL file written in C. Four incremental releases of Component C were analyzed and compared (henceforth referred to as Release C1 through Release C6, respectively). Each Component C release contains a DLL file with size of about 110 kilobytes.

The fourth case study (henceforth called Case 4) was conducted on a 405 KLOC ABB application (henceforth called Application D) written in C/C++. Application D incorporates 115 internal ABB software components of 104 `.dll` and 11 `.ocx` files written in C/C++. Four components were selected for study (henceforth referred to as Component D1 through Release D4, respectively). Each component is a Component Object Model (COM) [83] component, where D1, D2, and D3 are DLL components and D4 is an OCX component.

The COM is a software architecture that allows the components made by different software vendors to be combined into a variety of applications. COM defines a standard for component interoperability, is not dependent on any particular programming language, is available on multiple platforms, and is extensible. [83] The same four functions appear in the exports table of each COM component:

`DllCanUnloadNow`, `DllGetClassObject`, `DllRegisterServer`, and `DllUnregisterServer`. These functions are used to register/unregister the COM component, and are not the real interface functions that called by external systems. It is difficult to automatically find the component interface functions that can be called by external systems within the binary code of COM components. We have to first use Microsoft's OLE/COM Object Viewer (`oleview.exe`) [54] to extract the `.idl` file for the COM component. An *interface description language* (or *interface definition language*) (IDL) is a computer language used to describe a software component's interface in a language-neutral way, enabling communication between software components that do not share a language. Then we use the IDA Pro[9] tool to identify the start virtual addresses of the component interface functions can be called by external systems, and then use the Pallino tool to generate and analyze call graph for the real interface functions with in the components.

The subjects examined in our case studies are summarized in Table 5.1.

Table 5.1: Summary of case study subjects

Case	Application	Component	Number of Releases
1	A: 757 KLOC	A: one 67 KLOC `.lib` file in C	6
2	B: 40 KLOC	B: eight `.lib` files in C, totally 300 KLOC	5
3	A: 757 KLOC	C: one 3 KLOC `.dll` file in C	4
4	D: 405 KLOC	D1: one 3.9 KLOC COM DLL in C/C++	4
		D2: one 2.5 KLOC COM DLL in C/C++	4
		D3: one 4.5 KLOC COM DLL in C/C++	4
		D4: one 1.8 KLOC COM DLL in C/C++	4

[9] A disassembler and debugger, http://www.datarescue.com/

These software combinations were chosen for these case studies because (1) the numbers of test cases for each function of the applications were available; (2) multiple releases of the components were available; and (3) the high cost of executing the retest-all strategy for such large projects demonstrates the potential value of achieving regression test reductions.

For Case 1, 2, and 3, the author of this dissertation was the analyzer and Brian Robinson from ABB was the verifier. The analyzer conducted the first six steps of the I-BACCI Version 4. The results of the identified changes for all comparisons and all call graphs for the components were preliminarily verified by the analyzer, using source code for the component to determine the accuracy of the analysis post hoc. Then, the verifier determined the numbers and percent reduction of the regression test cases needed, based on the list of all the affected glue code functions and the original test suite. The verifier also confirmed the efficacy of the RTS process by examining the failure records of retest-all black-box testing. For Case 4, the analyzer first examined defect reports to analyze the association between defects and components to ensure that regression failures exist for the component/release. This step was conducted to avoid the situation that there is no regression failure found so that we could not obtain support for the effectiveness of the I-BACCI process through this case study. Four components that are associated with defects were selected for this case study. The analyzer and verifier then collaboratively conducted the I-BACCI process as have done in the first three case studies.

The rest of this Chapter presents all four case studies with the I-BACCI Version 4, which includes the Pallino tool discussed in Chapter 4. The effectiveness of the I-BACCI process based upon an examination of the failure records of retest-all black-box testing is reported as well. The analysis of total time costs for different RTS strategies for each release of the first three case studies are shown in Chapter 5.5. Additionally, Chapter 5.6 discusses the fifth case study that was conducted to evaluate the total time costs for regression testing of firewall analysis RTS technique by comparing with the modified entity RTS technique.

5.1 RESULTS OF CASE 1

The results of applying the I-BACCI Version 4 on Case 1 (Application A and Component A) are shown in Table 5.2. The interface between Application A's glue code functions and Component A was examined, to establish a baseline of affected functions in the application. In total, 60 functions (in 50 C++ files) in Application A call 89 functions of Component A. In the worst case, all 60 Application A functions would be affected by the changes in the Component A and would need to be re-tested.

The first analysis was conducted between Release A1 and Release A2 of Component A. The analysis showed that 18 functions were changed out of the 941 functions in Release A2, including three new functions. However, firewall analysis showed that 319 exported functions in Component A were affected by the identified changes. All 60 functions in Application A were affected. As a result, there was no regression test case reduction.

Table 5.2: Case 1 results by the I-BACCI Version 4

Metrics	Release Comparisons				
	1 vs 2	**2 vs 3**	**3 vs 4**	**4 vs 5**	**5 vs 6**
Total changed functions identified	18	23	1	10	3
True positive ratio[10]	100%	100%	100%	100%	100%
False positive ratio[11]	5.6 %	0 %	0 %	0 %	0%
Affected exported component functions	319	71	2	55	39
% of affected exported component functions	96.4%	21.5%	0.6%	16.6%	11.8%
Affected glue code functions	60	2	0	0	0
% of affected glue code functions	100%	3.3%	0%	0%	0%
Total test cases needed	592	8	0	0	0
% of test cases reduction	0%	98.7%	100%	100%	100%
Actual regression failures found	0	0	0	0	0
Regression failures detected by reduced test suite	0	0	0	0	0

The second analysis correctly identified 23 changed component functions, and 71 exported functions in the component were affected by the identified changes. Only two glue code functions called the affected exported component functions. Therefore, 98.7% regression test case reduction was achieved.

The latter three analyses identified only a few changes in the components and no function in Application A called any affected functions in the components, although the changes did affect some exported functions in the components. Therefore, 100% regression test case reduction for these three comparisons was achieved.

After all analysis was complete, the verifier examined the failure records of retest-all black-box testing. There were no regression test failures found. Therefore,

[10] True positives ratio is number of real changed functions found divided by total number of real changed functions.

[11] False positive ratio is number of identified changed functions that are not really changes, divided by number of (correctly and incorrectly) identified changed functions.

we could not obtain support for the effectiveness of the I-BACCI process through this case study. In the future case studies, we were sure to select case studies that contained regression test failures.

Additionally, the postmortem source code difference analysis showed that the change identification was correct except that one false positive existed in the comparison between Release A1 and Release A2. No false negative was found in the analyses.

5.2 RESULTS OF CASE 2

Similarly, the results of applying the I-BACCI Version 4 on Case 2 (Application B and Component B) are shown in Table 5.3. The interfaces between Application B's glue code functions and Component B were also examined to establish a baseline of affected functions in the application. The glue code functions changed in Release B3 of the application. For the former version of glue code functions (i.e. in Release 2), there were 123 exported functions in the component. In total, 46 glue code functions (in six C files) called 81 out of the 123 exported functions of the component. In the worst case, all of the 46 functions would be affected by the changes in the component and would need to be re-tested. Similarly, at most 59 glue code functions in the latter version would be affected.

The first analysis was conducted between Release B1 and Release B2 of the component. The BACCI analysis showed that 388 functions were changed out of 1,143 functions in Release B2. Firewall analysis showed that 84 exported functions in

Component B were affected by the identified changes and 38 glue code functions were affected. As a result, 30% of the regression test cases can be reduced.

Table 5.3: Case 2 results by the I-BACCI Version 4

Metrics	Release Comparisons			
	1 vs 2	**2 vs 3**	**3 vs 4**	**4 vs 5**
Total changed functions identified	338	1,238	4	13
True positive ratio	99.5%	98.4%	100%	100%
False positive ratio	4.9%	6.1%	0%	7.7%
Affected exported component functions	84	122	1	8
% of affected exported component functions	68.3%	100%	0.8%	6.6%
Affected glue code functions	38	59	1	6
% of affected glue code functions	82.6%	100%	1.7%	10.7%
Total test cases needed	151	215	11	20
% of test cases reduction	30%	0%	95%	91%
Actual regression failures found	4	8	1	0
Regression failures detected by reduced test suite	4	8	1	0

More reduction was achieved in the latter two comparisons (Release B3 vs. B4; Release B4 vs. B5): only 5% and 9% of the test cases needed to be re-run. However, due to the great extent of changes between Release B2 and B3, no regression test case reduction was found. Examination on the failure records of retest-all black-box testing supported the effectiveness of the I-BACCI Version 4 process as all 13 regression test failures would still be detected by the reduced regression test suite.

In the second case study, source code difference analysis showed that the tool was able to reduce false positives to only 6% on average while still having a low false negative rate (about 1%). The false negatives were caused because a changed function contained a function call which was replaced by another function call. The current tool ignored the address changes of the changed function call.

5.3 RESULTS OF CASE 3

The results of applying the I-BACCI Version 4 on Case 3 are shown in Table 5.4. To establish a baseline of affected functions in the application, the interface between Application A's glue code functions and Component C was examined. Only two functions in Application A call four functions of Component C.

The first analysis was conducted between Release C1 and Release C2. Of the 49 exported component functions in Release C2, 45 were changed. Both of the glue code functions in Application A that called Component C were affected. As a result, there was no regression test case reduction. The current tools identified all changes but had significant false positives when two versions were not built by the same linker.

Table 5.4: Case 3 results by the I-BACCI Version 4

Metrics	Comparisons		
	1 vs 2	2 vs 3	3 vs 4
Same linker?	No	Yes	Yes
Affected exported component functions	45	9	44
True positive ratio	100%	100%	100%
False positive ratio	60%	0%	0%
% of affected exported component functions	91.8%	18.4%	84.6%
Affected glue code functions	2	0	2
% of affected glue code functions	100%	0%	100%
Total test cases needed	31	0	31
% of test cases reduction	0%	100%	0%
Actual regression failures found	1	0	0
Regression failures detected by reduced test suite	1	0	0

The second analysis comparing Release C2 and Release C3 correctly identified nine affected exported component functions, but no function in Application A called any affected functions in the components. Therefore, we achieved 100% regression test case reduction for this comparison.

The result of the third analysis between Release C3 and Release C4 was similar to that of the first analysis, but no false positives were identified because the current tools worked well when comparing two releases built by the same linker.

The verifier examined the failure records of retest-all black-box testing. One regression test failure was found in the first comparison. No false negatives were found in any analyses.

5.4 RESULTS OF CASE 4

The results of applying the I-BACCI Version 4 on Case 4 are shown in Table 5.5.

Table 5.5: Case 4 results by the I-BACCI Version 4

Case	Metrics	Comparisons		
		1 vs 2	2 vs 3	3 vs 4
4.1	Same linker?	Yes	Yes	Yes
	Affected exported component functions	0	0	1
	True positive ratio	100%	100%	100%
	False positive ratio	0%	0%	0%
	% of affected exported component functions	0%	0%	7.7%
	Total test cases needed	0	0	101
	% of test cases reduction	100%	100%	88.0%
	Actual regression failures found	0	0	1
	Regression failures detected by reduced test suite	0	0	1
4.2	Same linker?	Yes	Yes	Yes
	Affected exported component functions	0	0	3
	True positive ratio	100%	100%	100%
	False positive ratio	0%	0%	33.3%
	% of affected exported component functions	0%	0%	75%
	Total test cases needed	0	0	81
	% of test cases reduction	100%	100%	90.4%
	Actual regression failures found	0	0	1
	Regression failures detected by reduced test suite	0	0	1
4.3	Same linker?	Yes	Yes	Yes
	Affected exported component functions	0	10	0
	True positive ratio	100%	100%	100%
	False positive ratio	0%	90%	0%

	% of affected exported component functions	0%	66.7%	0%
	Total test cases needed	0	21	0
	% of test cases reduction	100%	97.6%	100%
	Actual regression failures found	0	1	0
	Regression failures detected by reduced test suite	0	1	0
4.4	Same linker?	Yes	Yes	Yes
	Affected exported component functions	3	0	0
	True positive ratio	100%	100%	100%
	False positive ratio	33.3%	0%	0%
	% of affected exported component functions	42.9%	0%	0%
	Total test cases needed	56	0	0
	% of test cases reduction	93.4%	100%	100%
	Actual regression failures found	1	0	0
	Regression failures detected by reduced test suite	1	0	0

Among the four sub case studies, no affected exported component functions were missed by Pallino, which means there was no any false negative. All modifications and their impact that associated with all the defects can be identified correctly.

5.5 TIME COSTS ANALYSIS

Pallino were run on an IBM T42 laptop with one Intel® Pentium® M 1.8 GHz processor and one gigabyte RAM. The comparisons of total time costs among different RTS strategies for each release of the first three case studies are shown in Table 5.6. The time cost analysis was not able to be conducted on Case 4 due to the lack of data in test execution time costs. One assumption is that the mapping of all test cases with the glue code functions they execute is ready. Also, another limitation is that we only have rough estimation on time costs of test execution.

The range of values in test execution was based upon the results of tests reduction of applying the I-BACCI process to the releases. When significant changes were identified in the new release and no test cases could be eliminated, the full test

execution cycle is necessary, there is no reduction compared to retest-all strategy, but an increase in analysis time. Conversely, when the changes in the component do not affect the application, or there is no change identified in the component, the test execution cycle is zero because no test cases need to be re-run. In this case, the total time cost for regression testing is merely the time cost of the BCA and RTS analysis.

Table 5.6: Rough total time costs

Case	Approach	Time Costs (for each release)			
		BCA	RTS	Test Execution	Total
1	Retest-all	0	0	1 month	1 month
	I-BACCI (manually)	5 days	2 hrs	0 ~ 1 month	5 days ~ 1.1 month
	I-BACCI (w/ Pallino)	2 mins	2 hrs	0 ~ 1 month	2 hrs ~ 1 month
2	Retest-all	0	0	5 months	5 months
	I-BACCI (manually)	15 days	1 hr	0 ~ 5 month	15 days ~ 5.5 month
	I-BACCI (w/ Pallino)	5 mins	1 hr	0 ~ 5 month	1 hr ~ 5 month
3	Retest-all	0	0	4 days	4 days
	I-BACCI (manually)	much more than 4 days	2 hrs	0 ~ 4 days	much more than 4 days
	I-BACCI (w/ Pallino)	15~19 mins	2 hrs	0 ~ 4 days	2.5 hrs ~ 4 days

Although there is no time costs in BCA and RTS, retest-all strategy takes a large amount of time in test execution. Conducting the I-BACCI process without automation can be time consuming as well, especially for analyzing PE components. For example, the BCA of Case 3 is very complex and it would take much more time than just retest all test cases in four days. With the help of Pallino, the I-BACCI process can be completed in about one to two person hours for each release of the case studies. Depending upon the percentage of test cases reduction determined by

the I-BACCI process, the total time cost of the whole regression testing process can be reduced from five person months by retest-all strategy to one person hour in the best case.

5.6 FIREWALL RTS VS. MODIFIED ENTITY RTS

Firewall analysis restricts regression testing to potentially-affected system elements that can be reached within one edge from changed system elements in the call graph [79, 81]. Although theoretically not safe, firewall analysis was shown to be effective and efficient via empirical studies of industrial real-time systems [81]. As the most efficient safe RTS technique, the modified entity technique selects all tests that can reach the modified functions in a software system at a course granularity level [14, 16, 63]. Firewall analysis is theoretically more efficient and precise than the modified entity by analyzing fewer functions in the programs and selecting fewer test cases, and therefore spending less time on the full regression testing cycle. We

The case studies discussed in Chapters 5.1-5.4 do not justify to what extent the firewall analysis RTS technique is more efficient than usage of other RTS techniques which are discussed in Chapter 2.2. To compare the efficiency of firewall analysis and the safe modified entity RTS techniques [64], another case study (henceforth called Case 5) was conducted on the source code of a 930 KLOC ABB application (henceforth called Application E) written in C/C++. Application E was selected for this case study because the test cases were automated integration test cases, such that modified entity technique can be applied for RTS.

Application E is composed of 47 interactive modules for which we had all source and executable code. To imitate the context of user-oriented component regression testing, two modules in the bottom of the module call chain were selected and regarded as "components." Then we identified changes from these components between two releases of Application E, and analyzed the change impact to the rest of the application. The firewall analysis and the modified entity RTS techniques were applied to select test cases for testing the components, respectively. The results are shown in Table 5.7.

For example, there are two changed functions in the "Component 1," and all functions within the "Component 1" are called through the two functions. Firewall analysis selects 521 test cases that execute the functions in the rest of Application E that directly call the two "exported component" functions. Modified entity technique selects 623 test cases that can reach the two "exported component" functions in the rest of Application E, which is a superset of the set of test cases selected by firewall analysis. Using firewall analysis and modified entity techniques achieved 87.6% and 85.1% reduction in the number of test cases, respectively. However, the manual analysis time with modified entity is three times more than that with firewall analysis, because the execution trace for all the functions in the system needs to be analyzed. The calling trace to the "Component 2" in Application E is simple and 17 test cases (0.4% of the number of all test cases) were selected by both firewall analysis and modified entity techniques. However, unfortunately we do not have the data of time

costs in running these selected test cases.

Table 5.7: Comparison of RTS techniques

Metrics	"Component"	
	1	**2**
Size of the "component" (KLOC)	4.6	5.3
Changed functions in the "component"	2	9
Affected "exported component" functions	2	12
Total test cases needed with I-BACCI firewall analysis	521	17
Total test cases needed with modified entity	623	17
Total test cases needed with retest-all	4193	4193
Analysis time with I-BACCI firewall analysis (manually)	4 hours	10 mins
Analysis time with modified entity (manually)	12 hours	10 mins

5.7 SUMMARY OF CASE STUDIES

Generally, the higher percentage of affected exported component functions, the lower the percentage of test cases reduction, as shown in Figure 5.1. In the best case, as much as 100% regression test case reduction can be achieved by the I-BACCI process if our analysis indicates the changes to the COTS component are not called by the glue code. This fact would not be known to the users of COTS component without I-BACCI analysis, such that they would still be tempted to use the retest-all RTS method. When there are a large number of changes in the new release of the component, the I-BACCI process suggests a retest-all regression test strategy, similar to other RTS techniques. Also, the I-BACCI process is more effective when there are small incremental changes between revisions, as is true with all RTS techniques. The results were verified by examining the failure records of retest-all black-box testing. All regression test failures can be found by reduced test suites in the comparisons of

these case studies. No false negatives were found in any analyses. The time costs analysis showed the efficiency of the I-BACCI process. Depending upon the percentage of test cases reduction determined by the I-BACCI process, the total time cost of the whole regression testing process can be reduced to 0.0003% of that by retest-all strategy (from five person months by retest-all strategy to one person hour by the I-BACCI process) in the best case.

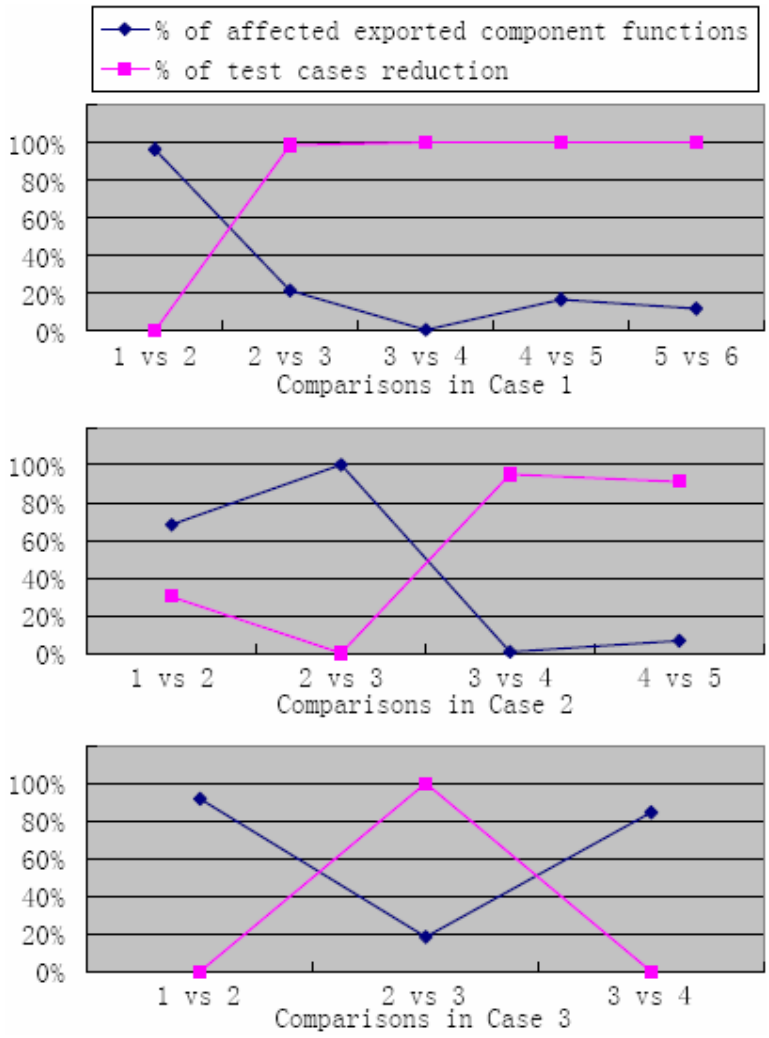


Figure 5.1: Relationship between the percentage of affected exported component functions and the percentage of test cases reduction for each case study

A limitation of the case studies is that all of the applications and components used were software developed by ABB Inc. involving `.lib` and `.dll` library files. Additionally, the Pallino tool currently works well only when the releases of components are built by the same linker. If two compared releases are built by different compilers or linkers, the current tools used in the I-BACCI process will yield a significant number of false positives. Also, we do not yet have accurate data on the saving of regression testing time for the case studies.

CHAPTER 6

LEGAL ISSUES

This chapter discusses the legal issues related to this research. This chapter is based upon a paper [39] that was written in conjunction with Dr. Cem Kaner, a lawyer and software engineering professor. Several of the paragraphs in this chapter were written by Dr. Kaner in the process of writing that paper. Dr. Kaner's legal writing is included in this chapter with his permission.

6.1 PROBLEM MOTIVATION

Software developers generally write source code in high level programming languages. Compilers translate that source into binary code that computer processors can recognize and execute. During maintenance, the development team analyzes the source or binary code to determine the next steps. For example, developers can use their knowledge of what was changed — at the function/method and line of code level — to determine what to retest before releasing the evolved software to customers.

Purchasers of commercial-off-the-shelf (COTS) components, unfortunately, must perform maintenance activities “in the dark” since they are not provided source code and analyzing binary code can be considered reverse engineering and illegal, as software licenses contain broad bans on reverse engineering. However, to enforce a

reverse engineering ban can impose unnecessary costs on a COTS customer without realizing corresponding benefits to the COTS vendor. As will be shown in this chapter, I-BACCI is a reverse engineering activity that involves analysis of a software product's binary code but that poses absolutely no competitive threat to the software publisher.

6.2 REVERSE ENGINEERING OF SOFTWARE

Twenty-eight software license agreements were gathered to investigate the legality of analyzing binary code of purchased COTS components. Relevant sentences in the license agreements were reviewed by lawyers of North Carolina State University (NCSU). Many of these license agreements of commercial components prohibit the users of components from reverse engineering, decompiling, disassembling, or otherwise attempting to discover the source code of the software, except to the extent that this restriction is expressly prohibited by law. Copyright law does not prohibit analysis on the code, only prohibits reproducing the components, making derivative works, or distributing copies of the products. The purpose of this research is to reduce the testing required when components change and only binary code and documentation is available. As a result, the NCSU lawyers deemed that the approaches and algorithms used in the I-BACCI process legal due to the purpose of the analysis.

In their definitive paper, Chikovsky and Cross [17, p. 15] defined reverse engineering as "the process of analyzing a subject system to identify the system's

components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." Reverse engineering is commonplace in all types of product development and maintenance for many reasons, such as learning how to use a product, assessing its safety, determining whether it can meet advertised claims, figuring out how to fix it, academic research, and of course, figuring out how to build a better product to compete with this one.

Courts in the United States routinely rule that reverse engineering is a lawful activity. The rationale is that, even though reverse engineering can reveal underlying inventions that make a new product work, the way to protect those inventions is to patent them. The inventor reveals the patented technology to the public; in return, no one can use it without the patent holder's permission (license).

Software reverse engineering is more complex because the process of reverse engineering typically involves making copies of the software or modifications (derivative works), such as translations from one language (e.g. machine language) to another (higher level) language, which is what disassembly and decompilation actually do. Patent law protects the ideas in the software, but copyright law protects the expression — what the code says if you read it, and what the program displays or transmits. Under copyright law, only the copyright holder and her or his licensees may make copies or derivatives of a work. It might seem, therefore, that software reverse engineering is not lawful without permission.

American courts have repeatedly ruled that software reverse engineering is

perfectly legal because the Copyright Act allows certain types of copying without permission. Collectively, these are called *fair use*. Photocopying part of a book for classroom teaching is an example of fair use. So is quoting sections of a book in a critical review of that book. Software reverse engineering fits this category too. The reason the copies are made is not to have more copies. It is to get at the underlying ideas — these cannot be protected by copyright law, only by patent. Therefore, the courts rule, nothing that the Copyright Act should protect is being infringed; making these temporary copies is fair use.

If software was sold the way books are sold, the story would stop here. But software is rarely sold; it is licensed. A license is a contract in which the holder of an intellectual property (IP) right grants some type of IP-related permission to the licensee. When you load software from a disk to a computer's memory, you make a copy — there is one copy on the disk, one in RAM. It is the license that grants the permission to make that copy. Licenses often include restrictions as well as permissions. One common restriction prohibits reverse engineering.

A recent case [3] illustrates exactly what the software publishers are trying to protect themselves from — one company apparently reverse engineered the software of another to create a competing product that looked and worked just like the original. Presumably, the copying company's R&D costs were much lower than the original publisher's, so allowing this clone product on the market could be grossly unfair. The original software came with a standard-form contract that barred all forms of reverse

engineering. The court upheld this restriction. A reader who considered only the facts of this particular case might consider that decision wise and just. However, it has broader implications that could constrain how most software developers work.

Bowers v. Baystate Technologies [3, p. 1326], defined reverse engineering very broadly: "to study or analyze (a device, as a microchip for computers) to learn details of design, construction, and operation, perhaps to produce a copy or an improved version." This is consistent with Chikovsky and Cross's definition. This goes far beyond disassembly or decompilation. You might study (reverse engineer) a product by testing it or even by analyzing its documentation. Indeed, IEEE Standard 1012 specifically recommends deriving a system's requirements and design by reverse engineering them (if necessary) from the user's manual. If a broad ban on reverse engineering is enforceable, much of what we do when we develop, maintain, or study code apparently cannot lawfully be done.

The fact that a restriction appears in a contract does not necessarily make it enforceable under American law. Courts have authority to strike any term that violates public policy. Some have done this to preserve the ability to reverse engineer to achieve interoperability. There are ongoing arguments for protecting software reverse engineering should be protected for a much broader set of activities [43].

The law of software licensing is enormously complex. The American Law Institute (ALI) and the National Conference of Commissioners on Uniform State Laws (NCCUSL) are the two leading private organizations who guide American legislatures

and judges in the drafting and interpretation of complex laws. For example, they co-authored the *Uniform Commercial Code*, which is the bedrock of commercial law in all 50 states. They worked together to try to draft a uniform law of software licensing for nearly 10 years, eventually disagreeing so vigorously that ALI abandoned the project, and NCCUSL published the *Uniform Computer Information Transactions Act* (UCITA) on its own in 2000. In state legislatures, UCITA failed. Only Virginia and Maryland adopted it, creating special rules for those two states different from the panoply of other rules in all the other states. Five years later, ALI started another project to draft a *Principles of the Law of Software Contracts*. It is likely that American law governing the reverse engineering clauses in software contracts will stabilize around the results of this project. The ALI welcomes comments on its drafts; it would make sense for the computing community to publish detailed examples of how we use reverse engineering, explaining why (and when) the particular example should be protected or blocked by contract.

6.3 LEGAL LIMITS OF BCA OF PURCHASED SOFTWARE

The I-BACCI process is not used to create the COTS components. The use described here, which we expect to be the normal application of I-BACCI, is for reducing time and cost of regression testing in the face of changes in COTS components. The I-BACCI process promotes the cost-effective continuation of interoperability between the customer's product and the COTS components purchased to build the product.

To study the legality of analyzing binary code of purchased COTS components, we gathered twenty-eight software license agreements. Many commercial component licenses prohibited component users from reverse engineering, decompiling, disassembling, or otherwise attempting to discover the source code of the software, except to the extent that this restriction is expressly prohibited by law. Under the *Bowers* decision, the I-BACCI process might be blocked for use on most COTS components.

Such a restriction is probably prohibited under the European Community's Council Directive on the Legal Protection of Computer Programs, which expressly permits black box analysis of a software product by any legal possessor and decompilation when it is indispensable for achieving interoperability with another program.[28]

In the United States, UCITA was revised to include in its final (2002) version, Section 118 "Terms Relating to Interoperability and Reverse Engineering" to specifically permit reverse engineering to achieve interoperability, making contract terms that block this unenforceable. Achieving interoperability is also expressly favored in the *Digital Millennium Copyright Act* (United States Code Title 17 Section 1201(f)).

6.4 SUMMARY

The results of our I-BACCI case studies indicate that customers of COTS components can benefit greatly from change information compared to the black box

"guess what we changed" situation of today. Allowing these customers to use the I-BACCI process therefore makes the COTS components more affordable to the customer without costing the vendor a dime. We see no good reason to enforce blanket bans on reverse engineering in a way that restricts application such as the I-BACCI process because the ban is protecting the vendor against an entirely different risk.

In states that have adopted UCITA, *Bowers* should not operate to restrict the I-BACCI process' reverse engineering because it is done to achieve interoperability. In the other states, the question is still open.

Purchasers of such software would do well to contact their vendors and request waivers that allow them to reverse engineer COTS components for the purpose of managing their maintenance costs. Vendors of these components would serve their customers well by revising their licenses to specifically permit this kind of analysis.

CHAPTER 7

CONTRIBUTIONS AND FUTURE WORK

More and more COTS components are incorporated in software products. Industry desires an effective and efficient approach of selecting regression tests when the COTS components included in their applications change. However, the majority of existing RTS techniques rely on source code for change identification and impact analysis, and therefore are not suitable when source code is not available for analysis. In this dissertation, we have present the application of the I-BACCI process that reduces the regression test suite using the firewall analysis RTS method based upon static change identification and impact analysis in the binary code of the COTS component. We also present Pallino, a tool that statically identifies binary code changes and their impact to support regression test selection for COTS-based applications when source code of components is not available. Pallino was designed to support the I-BACCI process but could be extended and/or modified to support other RTS methods for COTS components when source code is not available. Pallino can be applied to binary files of components in either COFF or PE format written in C/C++ at this stage.

Five case studies were conducted at ABB on products written in C/C++. The results showed that the I-BACCI process is an effective RTS process for COTS-based applications. The I-BACCI process can reduce the required number of regression test by as much as 100% if there are a small number of changes in the new component. Similar to other RTS techniques, when there are a large number of changes in the new component, I-BACCI suggests a retest-all regression testing strategy. The results of the case studies had been verified by examining the failure records of retest-all black-box testing. No failures would have escaped the reduced test suite. The I-BACCI process can be most beneficial when there are small incremental changes between revisions. With the help of Pallino, the I-BACCI process can be completed in about one to two person hours for each case study. Depending upon the percentage of test cases reduction determined by the I-BACCI process, the total time cost of the whole regression testing process can be reduced to 0.0003% of that by retest-all strategy in the best case.

These results supported the theory we are building:

When components change and source code is not available, regression tests can be selected from the test cases that execute the glue code that is in the call chain of functions of the component that changed, with minimal reduction in regression fault detection ability.

The main contributions of this dissertation are:

- *Development of an effective, efficient, and overall less expensive RTS*

solution for COTS-based applications when source code of components is not available.

- *Empirical evidence of the ability of the I-BACCI process to reduce the regression test cases required using case studies in industrial environments.*
- *Combination of BCA and firewall analysis technique. The I-BACCI process extended the traditional concept and scope of application for firewall analysis for use with binary code.*
- *An open source supporting tool that statically identifies binary code changes and their impact to support regression test selection for COTS-based applications when source code of components is not available. Pallino can efficiently identify affected exported component functions, and therefore facilitate reducing the required number of regression test.*
- *Investigation of the legal issues related to reverse engineering of software and the limits of BCA of purchased software.*

We will pursue several directions in our future work. Besides expanding the I-BACCI process to adapt to more programming languages and more of the COTS types, such as components in the ELF format, we plan to address the limitations of the I-BACCI process which are discussed in Chapter 3.4. We will reduce as many false positives as possible caused by factors other than source code (e.g. build tools, environment, and target platforms). Additionally, extensive validation of both the tool support and RTS process will require more industrial case studies, data collection, and

further RTS analysis.

REFERENCES

[1] The American Law Instituter, http://www.ali.org/

[2] The National Conference of Commissioners on Uniform State Lawsr, http://www.nccusl.org/

[3] "Bowers v. Baystate Technologies," in *Federal Reporter 3d.* vol. 320: United States Court of Appeals for the Federal Circuit, 2003, p. 1317.

[4] K. Abdullah, J. Kimble, and L. White, "Correcting for Unreliable Regression Integration Testing," in *International Conference on Software Maintenance*, Nice, France, 1995, pp. 232-241.

[5] H. Agrawal, J. Horgan, E. Krauser, and S. London, "Incremental Regression Testing," in *Conference on Software Maintenance*, September 1993.

[6] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A Differencing Algorithm for Object-Oriented Programs," in *19th International Conference on Automated Software Engineering (ASE'04)*, Linz, Austria, 2004, pp. 2-13.

[7] R. Arnold and S. Bohner, *Software Change Impact Analysis*: Wiley-IEEE Computer Society Press, 1996.

[8] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "CodeSurfer/x86 -- A platform for analyzing x86 executables," in *International Conference on Compiler Construction*, April 2005.

[9] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum, "WYSINWYX: What You See Is Not What You eXecute," in *The IFIP Working Conference on*

Verified Software: Theories, Tools, Experiments, 2005.

[10] T. Ball, "On the Limit of Control Flow Analysis for Regression Test Selection," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, Clearwater Beach, FL, March 1998.

[11] S. Bates and S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," in *20th ACM Symposium on Principles of Programming Languages*, January 1993, pp. 384-396.

[12] P. Benedusi, A. Cimitile, and U. D. Carlini, "Post-Maintenance Testing Based on Path Change Analysis," in *Conference on Software Maintenance*, October 1988, pp. 352-361.

[13] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs," in *The International Symposium on Requirements Engineering for Information Security*, 2001, pp. 1-8.

[14] J. Bible, G. Rothermel, and D. Rosenblum, "A Comparative Study of Course- and Fine-Grained Safe Regression Test-Selection Techniques," *ACM Transactions on Software Engineering and Methodology,* vol. 10(2), pp. 149-183, 2001.

[15] D. Binkley, "Reducing the cost of Regression Testing by Semantics Guided Test Case Selection," in *International Conference on Software Maintenance*, October 1995, pp. 251-260.

[16] Y. F. Chen, D. S. Rosenblum, and K. P. Vo, "TestTube: A System for Selective Regression Testing," in *16th International Conference on Software Engineering*, May 1994, pp. 211-222.

[17] E. Chikovsky, J. and J. H. Cross, II, " Reverse engineering and design recovery: a taxonomy," *IEEE Software,* vol. 7, pp. 13-17, Jan. 2000.

[18] M. Christodorescu, N. Kidd, and W. Goh, "String analysis for x86 binaries," in *Workshop on Program Analysis For Software Tools and Engineering*, Lisbon, Portugal, September 2005.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition ed. Cambridge , Massachusetts London, England: The MIT Press and McGraw-Hill, 2001.

[20] 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, pp. 99-123, 2001.

[21] K. F. Fischer, "A Test Case Selection Method for the Validation of Software Maintenance Modifications," in *International Computer Software and Applications Conference*, November 1977, pp. 421-426.

[22] K. F. Fischer, F. Raji, and A. Chruscicki, "A Methodology for Retesting Modified Software," in *National Telecommunications Conference*, November 1981, pp. 1-6.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley Professional, 1995.

[24] J. Gao and Y. Wu, "Testing Component-Based Software - Issues, Challenges, and Solutions," in *3rd International Conference on COTS-Based Software Systems*, Redondo Beach, 2004.

[25] J. Z. Gao, H.-S. J. Tsao, and Y. Wu, *Testing and Quality Assurance for Component-Based Software*. Boston: Artech House, 2003.

[26] T. L. Graves, M. J. Harrold, Y. M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," *ACM Transactions on Software Engineering and Methodology,* vol. 10(2), pp. 184-208, 2001.

[27] S. Gregor, "The nature of theory in information systems," *Management Information Systems Quarterly,* vol. 30, p. 611¨C642, 2006.

[28] C. M. Guillou, "The reverse engineering of computer software in Europe and the United States: A comparative approach," *Columbia-VLA Journal of Law & the Arts,* vol. 22, pp. 533-556, 1998.

[29] R. Gupta, M. J. Harrold, and M. L. Soffa, "An Approach to Regression Testing Using Slicing," in *Conference on Software Maintenance*, November 1992, pp. 299-308.

[30] J. E. Hannay, D. I. K. Sjoberg, and T. Dyba, "A Systematic Review of Theory Use in Software Engineering Experiments," *IEEE Transactions on Software Engineering,* vol. 33, pp. 87-107, February 2007.

[31] M. J. Harrold, A. Orso, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do, "Using Component Metacontents to Support the Regression Testing of Component-Based Software," in *IEEE International Conference on Software Maintenance (ICSM 2001)*, Florence, Italy, 2001, pp. 716-725.

[32] M. J. Harrold and M. L. Soffa, "Interprocedural Data Flow Testing," in *Third Testing, Analysis, and Verification Symposium*, December 1989, pp. 158-167.

[33] M. J. Harrold and M. L. Soffa, "An Incremental Data Flow Testing Tool," in *Sixth International Conference on Testing Computer Software*, May 1989.

[34] M. J. Harrold and M. L. Soffa, "An Incremental Approach to Unit Testing During Maintenance," in *Conference on Software Maintenance*, October 1988, pp. 362-367.

[35] J. Hartmann and D. J. Robson, "RETEST-Development of a Selective Revalidation Prototype Environment for Use in Software Maintenance," in *23rd Hawaii International Conference on System Sciences*, January 1990, pp. 92-101.

[36] J. Hartmann and D. J. Robson, "Techniques for Selective Revalidation," *IEEE Software,* vol. 16, pp. 31-38, January 1990.

[37] J. Hartmann and D. J. Robson, "Revalidation During the Software Maintenance Phase," in *Conference on Sofiware Maintenance*, October 1989, pp. 70-79.

[38] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE*

Standard 610.12, 1990.

[39] C. Kaner, J. Zheng, L. Williams, B. Robinson, and K. Smiley, "Binary Code Analysis of Purchased Software: What are the Legal Limits?," *Submitted to the Communications of the ACM,* 2007.

[40] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change Impact Identification in Object-Oriented Software Maintenance," in *International Conference on Software Maintenance*, Victoria, B.C., Canada, 1994, pp. 202-211.

[41] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Class Firewall, Test Order and Regression Testing of Object-Oriented Programs," *Journal of Object-Oriented Programming,* vol. 8(2), pp. 51-65, 1995.

[42] J. Laski and W. Szermer, "Identification of Program Modifications and Its Applications in Software Maintenance," in *Conference on Software Maintenance*, November 1992, pp. 282-290.

[43] D. Laster, "The Secret Is Out: Patent Law Preempts Mass Market License Terms Barring Reverse Engineering for Interoperability Purposes," *Baylor Law Review,* vol. 58, pp. 621-706, 2006.

[44] J. A. N. Lee and X. He, "A Methodology for Test Selection," *The Journal of Systems and Software,* vol. 13, pp. 177-185, September 1990.

[45] H. Leung and L. White, "A Study of Integration Testing and Software Regression at the Integration Level," in *International Conference on Software*

Maintenance, San Diego, 1990, pp. 290-301.

[46] H. Leung and L. White, "Insights into Testing and Regression Testing Global Variables," *Journal of Software Maintenance,* vol. 2(4), pp. 209-222, 1991.

[47] H. Leung and L. White, "Insights into Testing and Regression Testing Global Variables," *Journal of Software Maintenance,* vol. 2, pp. 209-222, December 1991.

[48] H. Leung and L. White, "A Study of Integration Testing and Software Regression at the Integration Level," in *Conference on Software Maintenance*, San Diego, November 1990, pp. 290-300.

[49] L. Mariani, S. Papagiannakis, and M. Pezze, "Compatibility and regression testing of COTS-component-based software," in *29th International Conference on Software Engineering*, Minneapolis, MN, 2007, pp. 85-95.

[50] L. Mariani and M. Pezze, "Behavior capture and test: Automated analysis of component integration," in *International Conference on Engineering of Complex Computer Systems*, 2005.

[51] A. v. Mayrhauser, R. T. Mraz, and J. Walls, "Domain Based Regression Testing," in *International Conference on Software Maintenance*, 1994, pp. 26-35.

[52] A. M. Memon, "A process and role-based taxonomy of techniques to make testable COTS components," in *Testing Commercial-off-the-shelf Components and Systems*, S. Beydeda and V. Gruhn, Eds. Berlin, Germany:

Springer-Verlag, 2005, pp. 109-140.

[53] Microsoft, "DUMPBIN Reference," in *MSDN Library*.

[54] Microsoft, "Using OleView," in *MSDN Library*.

[55] Microsoft, "Common Object File Format (COFF)," in *MSDN Library*, 2001.

[56] A. Orso, R. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, 2004, pp. 491-500.

[57] T. J. Ostrand and E. J. Weyuker, "Using Dataflow Analysis for Regression Testing," in *Sixth Annual Pacific Northwest Software Quality Conference*, September 1988, pp. 233-247.

[58] M. Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format," in *MSDN Library*, March 1994.

[59] M. Pietrek, "An In-Depth Look into the Win32 Portable Executable File Format," in *MSDN Magazine*, March 2002.

[60] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip, "Chianti: A Change Impact Analysis Tool for Java Programs," in *the 27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005, pp. 664-665.

[61] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, and J. Dolby, "Chianti: A prototype change impact analysis tool for Java," Technical Report DCS-TR-533: Department of Computer Science, Rutgers University,

September 2003.

[62] G. Rothermel, "Efficient, Effective Regression Testing Using Safe Test Selection Techniques," in *PhD dissertation*: Clemson University, May 1996.

[63] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," *IEEE Trans. on Software Engineering,* vol. 22(8), pp. 529-551, 1996.

[64] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 6, pp. 173 - 210, 1997.

[65] G. Rothermel and M. J. Harrold, "Selecting Tests and Identifying Test Coverage Requirements for Modified Software," in *International Symposium on Software Testing and Analysis*, August 1994, pp. 169-184.

[66] G. Rothermel and M. J. Harrold, "A Safe, Efficient Algorithm for Regression Test Selection," in *Conference on Software Maintenance*, September 1993, pp. 358-367.

[67] G. Rothermel and M. J. Harrold, "Selecting Regression Tests for Object-Oriented Software," in *International Conference on Software Maintenance*, September 1994, pp. 14-25.

[68] B. Sherlund and B. Korel, "Logical Modification Oriented Software Testing," in *22th International Conference on Testing Computer Software*, June 1995.

[69] B. Sherlund and E. Korel, "Modification Oriented Software Testing," *Quality*

Week 1991, pp. 1-17, 1991.

[70] A. Srivastava, "Vulcan," Microsoft Research TR-99-76, 1999.

[71] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, Roma, Italy, 2002, pp. 97-106.

[72] A. B. Taha, S. M. Thebaut, and S. S. Liu, "An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis," in *13th Annual International Computer Software and Applications Conference*, September 1989, pp. 527-534.

[73] F. Vokolos and P. Frankl, "Pythia: A regression test selection tool based on textual differencing," in *3rd International Conference on Reliability, Quality and Safety of Software-intensive System*, Athens, Greece, 1997, pp. 3-21.

[74] F. Vokolos and P. Frankl, "Empirical Evaluation of the Textual Differencing Regression Testing Technique," in *International Conference on Software Maintenance*, 1998, pp. 44-53.

[75] Z. Wang, K. Pierce, and S. McFarling, "BMAT: A Binary Matching Tool for Stale Profile Propagation," *The Journal of Instruction-Level Parallelism,* vol. Vol. 2, 2000.

[76] E. J. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software,* vol. 15(5), pp. 54-59, 1998.

[77] L. White and K. Abdullah, "A Firewall Approach for the Regression Testing of

Object-Oriented Software," in *Software Quality Week* San Francisco, 1997.

[78] L. White, H. Almezen, and S. Sastry, "Firewall Regression Testing of GUI Sequences and Their Interactions," in *International Conference on Software Maintenance*, Amsterdam, The Netherlands, 2003, pp. 398-409.

[79] L. White and H. Leung, "A Firewall Concept for both Control-Flow and Data Flow in Regression Integration Testing," in *International Conference on Software Maintenance*, Orlando, 1992, pp. 262-271.

[80] L. White and H. Leung, "A Firewall Concept for both Control-Flow and Data Flow in Regression Integration Testing," in *Conference on Software Maintenance*, November 1992, pp. 262-270.

[81] L. White and B. Robinson, "Industrial Real-Time Regression Testing and Analysis Using Firewall," in *International Conference on Software Maintenance*, Chicago, 2004, pp. 18-27.

[82] L. J. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha, "Test Manager: A Regression Testing Tool," in *Conference on Software Maintenance*, September 1993, pp. 338-347.

[83] S. Willliams and C. Kindel, "The Component Object Model: A Technical Overview," in *MSDN Library*, 1994.

[84] S. S. Yau and Z. Kishimoto, "A Method for Revalidating Modified Programs in the Maintenance Phase," in *21th Annual International Computer Software and Applications Conference*, October 1987, pp. 272-277.

[85] J. Zheng, B. Robinson, L. Williams, and K. Smiley, "A Lightweight Process for Change Identification and Regression Test Selection in Using COTS Components," in *5th International Conference on COTS-Based Software Systems*, Orlando, FL, USA, February 2006, pp. 137-143.

[86] J. Zheng, B. Robinson, L. Williams, and K. Smiley, "Applying Regression Test Selection for COTS-based Applications," in *28th IEEE International Conference on Software Engineering (ICSE'06)*, Shanghai, P. R. China, May 2006, pp. 512-521.

[87] J. Zheng, B. Robinson, L. Williams, and K. Smiley, "An Initial Study of a Lightweight Process for Change Identification and Regression Test Selection When Source Code is Not Available," in *16th IEEE International Symposium on Software Reliability Engineering*, Chicago, IL, USA, November 2005, pp. 225-234.

[88] J. Zheng, L. Williams, and B. Robinson, "Pallino: Automation to Support Regression Test Selection for COTS-based Applications," NCSU Technical Report, TR-2007-18, June 2007.

APPENDIX A

BINARY CODE COMPARISON FALSE POSITIVE PATTERNS

When comparing two versions of binary files for a COFF component, a large number of false positives were observed in the initial case study of the I-BACCI Version 1 [87]. A large amount of false positives were caused by changes in registers used and addresses of variables and functions, which typically would not cause functional changes in the code. For example, as shown in bold in the Figure 4.11, the binary code `8B89A0060000` means "copy the operand in the address of register ECX plus offset `0x06A0` to register ECX", where `8B89` is the opcode of the instruction and `A0060000` is the address offset. Therefore, in this example, the only difference in binary is that the address offset was changed from `A0060000` to `CC060000`. Further examination of the source code showed that seven new function declarations and one new variable definition were added before the variable state was defined in one of the header files included in the source file of the new release. As a result, the offset of the variable state was changed accordingly. In this case, the binary code change identified is not a real change and can be ignored in the change identification. The binary code like `8B89A0060000` is called an example of a

"*binary code comparison false positive pattern*." Many such false positive patterns were found in the first case study. The full list of these empirical patterns is shown in the Table A.1. False positive patterns are identified by their prefix. The prefix of a pattern can be the opcode of an instruction (e.g., `FF50`), or first few bits of an opcode (e.g., `8B8`) which means all opcode that begins with these bits are prefixes of false positive patterns (e.g., from `8B80` to `8B8F`). The algorithm scans the two versions of raw binary code of a function. For each false positive pattern, when the prefix of the pattern is found, the corresponding numbers of bytes from the start byte of the prefix are marked as a constant symbol (e.g., "_") in the raw code. Only if the remaining bytes of the two versions of binary code are the same, the function is considered as unchanged. The algorithm reduced the false positive rate to less than 8% in the case studies [86].

Table A.1: The full list of binary code comparison false positive patterns

Prefix of patterns	Bytes ignored	Description of related opcode
0F BE 8	7	Move byte to doubleword, with sign-extension
0F BF 8	7	Move word to doubleword, with sign-extension
0F BF 9	7	Move word to doubleword, with sign-extension
66 39 8	7	Compare r16 with r/m16
66 39 9	7	Compare r16 with r/m16
66 39 A	7	Compare r16 with r/m16
66 39 B	7	Compare r16 with r/m16
66 3B 8	7	Compare r/m16 with r16
66 83 B	7	Compare imm8 with r/m16
66 89 8	7	Move r16 to r/m16
66 89 9	7	Move r16 to r/m16
66 89 A	7	Move r16 to r/m16
66 89 B	7	Move r16 to r/m16
66 8B 8	7	Move r/m16 to r16

66 8B 9	7	Move r/m16 to r16
66 8B A	7	Move r/m16 to r16
66 C7 8	7	Move imm16 to r/m16
39 B	6	Compare r32 with r/m32
C6 8	6	Move imm8 to r/m8
C7 8	6	Move imm32 to r/m32
D9 9E	6	Copy ST(0) to m32fp and pop register stack
DC 86	6	Divide m64fp by ST(0) and store result in ST(0)
DC A6	6	Divide m64fp by ST(0) and store result in ST(0)
DC AE	6	Divide m64fp by ST(0) and store result in ST(0)
DD 86	6	Copy ST(0) to m80fp and pop register stack
DD 9E	6	Copy ST(0) to m80fp and pop register stack
F6 81	6	Signed divide EDX:EAX by r/m32
FF 90	6	Jump near/far, absolute indirect
FF 92	6	Jump near/far, absolute indirect
FF 95	6	Jump near/far, absolute indirect
FF 96	6	Jump near/far, absolute indirect
31 81	6	r/m32 XOR r32
39 86	6	Compare r32 with r/m32
39 9E	6	Compare r32 with r/m32
39 AE	6	Compare r32 with r/m32
39 BE	6	Compare r32 with r/m32
3B BE	6	Compare r/m32 with r32
81 BE	6	Compare imm32 with r/m32
81 C	6	Compare imm32 with r/m32
83 8E	6	Compare imm8 with r/m32
83 BE	6	Compare imm8 with r/m32
83 F8	6	Compare imm8 with r/m32
88 8	6	Move r8 to r/m8
88 9	6	Move r8 to r/m8
89 8	6	Move r32 to r/m32
89 9	6	Move r32 to r/m32
89 A	6	Move r32 to r/m32
89 B	6	Move r32 to r/m32
8A 8	6	Move r/m8 to r8
8A 9	6	Move r/m8 to r8
8B 8	6	Move r/m32 to r32
8B 9	6	Move r/m32 to r32
8B A	6	Move r/m32 to r32
8B B	6	Move r/m32 to r32
8D 8	6	Store effective address for m in register r32

8D 9	6	Store effective address for m in register r32
8D A	6	Store effective address for m in register r32
8D B	6	Store effective address for m in register r32
66 83 7	5	Compare imm8 with r/m16
05	5	Add imm32 to EAX or add imm16 to AX
68	5	Push sign-extended imm8/16/32. Stack pointer is incremented by the size of stack pointer.
C6 44 24	5	Move imm8 to r/m8
83 79	4	Add sign-extended imm8 to r/m16 or r/m32
0F BE 4	4	Move byte to word, with sign-extension
0F BF 4	4	Move byte to word, with sign-extension
0F BF 5	4	Move byte to word, with sign-extension
66 8B 4	4	Move r/m16 to r16
66 8B 5	4	Move r/m16 to r16
83 7D	4	Compare imm8 with r/m16 or r/m32
83 3B	3	Compare imm8 with r/m16
C7 40	3	Move imm16 to r/m16
FF 50	3	Jump near/far, absolute indirect
FF 52	3	Jump near/far, absolute indirect
89 41	3	Move r16 to r/m16
89 48	3	Move r16 to r/m16
8B 46	3	Move r/m8 to r8
8D 4E	3	Store effective address for m in register r16
6A	2	Push sign-extended imm8