Augmenting static program analysis and design verification with pattern recognition

Sun, Ding

2014

Sun, D. (2014). Augmenting static program analysis and design verification with pattern recognition. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/61780>

<https://doi.org/10.32657/10356/61780>

**AUGMENTING STATIC PROGRAM ANALYSIS
AND DESIGN VERIFICATION
WITH PATTERN RECOGNITION**

SUN DING

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

2014

**AUGMENTING STATIC PROGRAM ANALYSIS
AND DESIGN VERIFICATION
WITH PATTERN RECOGNITION**

DING S.

SUN DING

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
In partial fulfilment of the requirement for the degree of
Doctor of Philosophy

2014

ACKNOWLEDGMENT

The life I lived as a PhD student in NTU is a journey to reinvent myself. I learnt a lot from this journey. It gives me great pleasure to recount them here. Foremost acknowledge must go to Prof. Tan Hee Beng Kuan, whose unwavering support and providing me much guidance in my research. He has guided the project from the start; he guided me in details and shared with me his insightful knowledge in combining program analysis techniques with empirical knowledge to address important problems in program analysis. I would also like to thank my previous supervisor Dr Lin Qingping. He has also provided me great guidance before leaving NTU for his new career. I would also like to thank our collaborators Prof. Lionel C. Briand from SnT Centre, University of Luxembourg; Prof. Zhang Hongyu from School of Software, Tsinghua University.

We would like to thank the Beijing THSoft InfoTech (THIT), JetBrains, HLMZ.org and JiangSu Celestvision from China for providing the industry systems for our experiments within this thesis.

Thanks go, finally, to my family and friends. For my lovely wife Guo Ji, without my wife's support and encouragement, I would not have come this far. For my parents and grandparents, they taught me to be unwavering and focused, which is the faith supporting me to complete this thesis. For all of other familiar members and friends, I am grateful for all their love, support and care throughout this journey.

TABLE OF CONTENTS

Acknowledgment	iii
Table of Contents	iv
Summary	vii
List of Figures	ix
List of Tables	xi
Chapter 1 INTRODUCTION	1
1.1 Motivation	1
1.1.1 Design Verification of Database Applications	1
1.1.2 Infeasible Path Detection.....	2
1.1.3 Buffer Overflow Detection.....	2
1.1.4 Buffer Overflow Removal	3
1.2 Objectives.....	4
1.3 Major Contributions	4
1.4 Thesis Organization	5
Chapter 2 BACKGROUND.....	7
2.1 Program analysis	7
2.1.1 Control flow analysis.....	7
2.1.2 Data flow analysis	10
2.2 Domain background.....	11
2.2.1 Database design verification in relational model.....	11
2.2.2 Infeasible paths.....	12
2.2.3 Buffer overflow vulnerability	13
2.3 Empirical approaches for Pattern Recognition.....	16
Chapter 3 DESIGN VERIFICATION FOR DATABASE APPLICATION THROUGH ANALYSING THEIR SUPPORT TO DATA LIFECYCLE	18
3.1 Data Lifecycle and Transaction Patterns for Supporting	18
3.2 Design Verification via the Transaction Patterns for Providing Data Lifecycle	20
3.3 Empirical Evaluation.....	25
3.4 Conclusion.....	27
Chapter 4 DETECTION OF INFEASIBLE PATHS USING PATTERNS WITH LIMITED SYMBOLIC EVALUATION	28
4.1 Infeasible Paths Characterization	28

4.2	Detection of Infeasible Paths Using Patterns with Limited Symbolic Evaluation (DIPLE)	34
4.2.1	How DIPLE Works	34
4.3	Empirical Evaluation.....	39
4.3.1	Objectives and Dependent Variables.....	39
4.3.2	Experiment Design and Analysis	39
4.3.3	Automation	40
4.3.4	Experiment Results.....	42
4.3.5	Threats to Validity	48
4.4	Applications	49
4.5	Conclusion.....	49
Chapter 5 DETECTION OF BUFFER OVERFLOW VULNERABILITIES IN C/C++ WITH PATTERN BASED RESTRICTED SYMBOLIC EVALUATION		51
5.1	Buffer Overflow Patterns with Symbolic Evaluation.....	52
5.1.1	Code patterns	52
5.1.2	Restricted Symbolic Evaluation	56
5.2	Buffer Overflow Detection with Code Patterns	59
5.3	Empirical Evaluation.....	62
5.3.1	Implementation.....	62
5.3.2	Experiment Results.....	63
5.4	Limitation and discussion.....	66
5.5	Conclusion.....	66
Chapter 6 AUTOMATIC REMOVAL OF BUFFER OVERFLOW VULNERABILITIES IN C/C++ PROGRAMS.....		68
6.1	The Proposed Approach	69
6.1.1	Preliminary for ABOR.....	69
6.1.2	Buffer Overflow Detection	72
6.1.3	Buffer Overflow Removal	74
6.1.4	An example of ABOR	78
6.2	Evaluation	79
6.2.1	Experiment Design	79
6.2.2	Experimental Results.....	79
6.3	Conclusion.....	84
Chapter 7 RELATED WORK		86
7.1	Detection of Infeasible Path	86
7.1.1	Approaches	86
7.1.2	Tools Implementation.....	93

7.2	Buffer Overflow Handling	96
7.2.1	Method of Static Analysis	96
7.2.2	Method of Runtime Defence	99
7.2.3	Method of Attack Prevention through Auto Patching	99
7.3	Database Design Verification.....	99
7.4	Comparison and Summary	100
Chapter 8 CONCLUSION AND RECOMMENDATIONS		103
8.1	Conclusion.....	103
8.2	Limitation of the study	105
8.3	Future work and recommendations	106
APPENDIX A – Results of Experiment in Chapter 4.....		109
APPENDIX B – Notation conventions		127
Author’s Publications.....		129
Bibliography		131

SUMMARY

Static analysis examines program code to reason over all possible behaviours that might arise at run time. Such reasoning with full soundness and precision is in general not possible, as there may be arbitrarily many different user inputs or states. To enable the reasoning, static analysis usually uses an abstraction model to abstract away some information to examine program code. Static program analysis plays a very important role in software testing, software verification, program slicing, error detection, and code performance optimization. However, the use of it in these areas still faces many problems due to the difficulty in finding a suitable abstraction model to base on.

Design patterns have been used to improve the reuse in software development. Some design patterns have also been used in the verification of design in general. However, there are no specific patterns proposed for the verification of designs of database applications that constitute a large proportion of software systems.

The aim of this thesis is to address the above-mentioned problems. Based on abstraction models established through empirical studies, this thesis proposes the following four approaches augmented with pattern recognition, each of which serves an important purpose:

- 1) **Database design verification:** A lot of software systems are database applications. Functionalities provided in these applications are very much associated with their databases. Hence, many designs for these applications can be inferred from their databases. However, most of the design and verification techniques do not take advantage from these important and fundamental characteristics that are widely applied across database applications. Based on these characteristics, we propose a design verification approach for database applications. The proposed verification is grounded on the fact that all data maintained in a database must be inserted, used and removed. These data may also require to be modified. Our evaluation finds that the proposed design verification is useful and effective in detecting essential functionalities that are not provided originally.
- 2) **Infeasible path detection:** Detection of infeasible program paths is a major problem that is particularly important in structural testing. Many techniques have been proposed to detect infeasible paths. These techniques share many commonalities and also have differences. We integrate, extend and generalize existing techniques to detect infeasible paths with limited symbolic evaluation more effectively and accurately. We generalize

and extend the existing techniques for detecting infeasible paths in the form of two patterns to characterize most infeasible paths in real system code. These two patterns are detectable through limited symbolic evaluation, which we describe in this paper. Based on this characterization, we formulate a method called DIPLE (**D**etection of **I**nfeasible Paths Using **P**atterns with **L**imited **S**ymbolic **E**valuation) to detect infeasible paths through recognizing the patterns. DIPLE first uses syntax analysis, to filter away those paths that cannot possibly comply with any of the two patterns, before applying a limited symbolic evaluation for a precise recognition of the patterns. We have conducted extensive experiments on Java source code from multiple and diverse systems. DIPLE was evaluated based on sets of paths selected randomly and according to commonly used path coverage criteria. In the experiments, DIPLE shows high accuracy and efficiency. Furthermore, DIPLE is more scalable to detect infeasible paths than exhaustively using symbolic evaluation.

- 3) **Buffer overflow detection:** when people maintaining C/C++ programs, buffer overflow vulnerability is among the top security threats. However, existing approaches still have many practical issues. Some may generate many false positives/negatives. Some may not be scalable for large. It is worthwhile to explore the balance between detection accuracy and scalability. Therefore, we propose a novel method to detect vulnerable buffer overflows through recognizing two general patterns. The pattern recognition applies light weighted syntax analysis and a restricted version of symbolic evaluation. It also only evaluates limited paths and predicates only. So it can save considerable resources. This mechanism significantly improves the scalability while not sacrificing the detection precision. The experiments demonstrate the scalability and efficiency of the proposed method.
- 4) **Buffer overflow removal:** Researchers have put effort in different directions to address buffer overflow, including (1) creating a run-time defence mechanism, (2) proposing effective detection methods or (3) automatically modifying the original program to remove the vulnerabilities. However, it lacks integrated methods that can ultimately removal buffer overflow vulnerabilities with high precision but low cost. Therefore, we propose **Automatic Buffer Overflow Repairing (ABOR)**, a framework that removes buffer overflow vulnerabilities from source code automatically. ABOR starts with buffer overflow detection, followed by an automated vulnerability removal stage. ABOR only patches identified code segments; thus it is an optimised solution that can eliminate buffer overflows while keeping a minimum runtime overhead. We have implemented ABOR and evaluated it against both benchmarks and industry applications. The experiment results prove ABOR's effectiveness in practice.

LIST OF FIGURES

Figure 2-1. Sample program with its CFG	9
Figure 2-2. An infeasible path.....	13
Figure 2-3. A sample code with exploited buffer overflow vulnerability	15
Figure 2-4. Memory arrangement of the sample code	16
Figure 3-1. The lifecycle of attribute and record type.....	19
Figure 3-2 Use case template and an example [139].....	25
Figure 4-1. Flag-based inhibitor pattern.....	30
Figure 4-2. First conflicting branch pattern.....	32
Figure 4-3. Second conflicting branch pattern	34
Figure 4-4. A framework algorithm to apply DIPLE	37
Figure 4-5. An algorithm to infer the infeasibility of a path	38
Figure 4-6. The structure of our prototype system	41
Figure 4-7. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in BP0 experiment on intra-paths	43
Figure 4-8. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in BP0 experiment on inter-paths	44
Figure 4-9. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in CtP2 experiment on intra-paths	44
Figure 4-10. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in CtP2 experiment on inter-paths	44
Figure 4-11. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in Random experiment on intra-paths	44
Figure 4-12. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in Random experiment on inter-paths	44
Figure 4-13. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in BP0 experiment on intra-paths	45
Figure 4-14. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in BP0 experiment on inter-paths	45
Figure 4-15. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in CtP2 experiment on intra-paths	45
Figure 4-16. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in CtP2 experiment on inter-paths	45

Figure 4-17. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in Random experiment on intra-paths	46
Figure 4-18. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in Random experiment on inter-paths	46
Figure 4-19. Reconciliation of infeasible paths detected in DIPLE and exhSymbEvaPhInf in BP0 experiment on intra-paths	47
Figure 4-20. Reconciliation of Infeasible Paths Detected in DIPLE and exhSymbEvaPhInf in BP0 experiment on inter-paths	47
Figure 4-21. Reconciliation of Infeasible Paths detected in DIPLE and exhSymbEvaPhInf in CtP2 experiment on intra-paths	47
Figure 4-22. Reconciliation of infeasible paths detected in DIPLE and exhSymbEvaPhInf in CtP2 experiment on inter-paths	47
Figure 4-23. Reconciliation of infeasible paths detected in DIPLE and exhSymbEvaPhInf in Random experiment on intra-paths.....	48
Figure 4-24. Reconciliation of infeasible paths detected in DIPLE and exhSymbEvaPhInf in Random experiment on inter-paths.....	48
Figure 5-1. Pattern of vulnerable syntax usage	53
Figure 5-2. Pattern of Vulnerable Element Accessing	54
Figure 5-3. Pattern of vulnerable input validation.....	56
Figure 5-4. Algorithm of pattern recognition.....	61
Figure 5-5. Implementation of the proposed method.....	63
Figure 6-1. Overview of ABOR.....	69
Figure 6-2. Vulnerability detection in ABOR	74
Figure 6-3. Program Transformation.....	77
Figure 6-4. Comparison with a “Search & Replace” method.....	82
Figure 6-5. Comparison with a “Bounds Checker” method.....	82
Figure 7-1. Infeasible path detection with data flow analysis	87
Figure 7-2. An overview of path-based constraint propagation approach	89
Figure 7-3. An illustration of the path-based constraint propagation approach	89

LIST OF TABLES

Table 3-1. Type of errors.....	22
Table 3-2. Statistics obtained from our case studies	26
Table 4-1. System for experiment	42
Table 5-1. Accuracy of Proposed Method on Benchmarks.....	65
Table 5-2. Scalability Comparison between Proposed Method and Marple on Benchmarks	66
Table 5-3. Accuracy of Proposed Method on Industrial Applications	66
Table 5-4. Scalability between Proposed Method and Marple on Industrial Applications	66
Table 6-1. Detail of ABOR rule table	71
Table 6-2. Update Intermediate variable during propagation.....	75
Table 6-3. Vulnerability removal in ABOR.....	76
Table 6-4. <i>removeVul</i> for different prevention rules	77
Table 6-5. An example of applying AROR.....	78
Table 6-6. Vulnerabilities removed by ABOR in benchmarks	80
Table 6-7. Vulnerabilities removed by ABOR in industry programs.....	81
Table 6-8. The time performance of ABOR.....	81
Table 6-9. Examples of errors when processing the industry prog.	84
Table 6-10. Accuracy and error analysis	84
Table 7-1. Useful Tools for Implementation.....	94
Tables A.1.1-C.2.4. Tables for results of experiment in Chapter 4.....	109-126
Figure B.1. General program analysis notations	127
Figure B.2. Additional notations used by Chapter 4	127
Figure B.3. Additional notations used by Chapter 5	128
Figure B.4. Additional notations used by Chapter 6	128

Chapter 1

INTRODUCTION

1.1 Motivation

Static analysis and design verification Ayewah, et al. [24] provide means to verify programs' runtime behaviours to guarantee qualified software. They are very important in many software engineering tasks, such as testing, error detection and optimization. However, one may still face difficulties in finding suitable abstraction model to base on. One research direction of resolving this problem is to augment static analysis or design verification with abstraction model found from empirical studies [134, 156].

Empirical knowledge (also known as empirical data or empirical evidence) is the data from observation or experimentation, to justify a belief in the truth or falsity of an empirical claim [142]. In other scientific research fields such as physics, medicines and engineering, researchers have adopted empirical knowledge to augment formal methods, optimise systems and adjust experiment results. For example, medical pioneers have explored the working mechanism of human body and applied empirical knowledge to cure illness. Researchers also have adopted empirical knowledge in software engineering [30, 31, 76, 144, 156, 161, 177]. There are some successes in using design patterns that are found from empirical studies of the reuse of design and coding experiences in software development [16, 78, 81]. Though database applications constitute a large proportion of software systems in the world, there is no design patterns specifically proposed to verify the design of these applications.

This thesis addresses the above-mentioned problems with an empirical study based approach: through abstraction models established found from empirical studies, we establish patterns to augment design verification and static analysis to addresses four important problems: (1) design verification of database applications; (2) detection of infeasible paths; (3) detection of buffer overflow vulnerability; and (4) removal of buffer overflow vulnerability.

1.1.1 Design Verification of Database Applications

In the digital era, database accounts for an indispensable proportion for enterprises [72, 142]. Database manages an organization's essential information and sustains the organization's daily operation and vigorous extension. Therefore, design and development database with good quality in a cost effective way is crucial [150].

In software development lifecycle, it is well-known that the cost increases exponentially versus the time moves when developers fail to recognize and eliminate errors in the early stage. The earlier we can detect errors and rectify them, the lower the cost we would pay. Therefore, it is important to verify those earlier stage deliverables such as design specification and database integrity. However, most methods for such verifications are either too general or consists of formal specification and abstraction that are difficult for comprehension and implementation [51].

1.1.2 Infeasible Path Detection

One key challenge in static program analysis is the existence of infeasible program path that cannot be exercised by any input [40, 75, 79, 125]. The detection of infeasible paths has long been recognized as a major problem. As typically a sizable proportion of program paths are infeasible, the effectiveness and accuracy of structured testing techniques can be significantly improved [92] if most of the infeasible paths can be detected through static analysis and automatically excluded. Recent work [34, 170] discussed and highlighted the importance of detecting infeasible paths when using structural testing criteria and its acute impact on definition-use dataflow testing. In a comprehensive survey of test coverage [180], the adverse impact of the inability to detect infeasible statements, branches and paths on the precision of structural testing criteria has also been discussed. In modern test data generators that generate test data based on path-oriented criteria through combining static and dynamic analysis [151, 162, 173], as discussed in [58], the detection of infeasible paths without extensive symbolic evaluation would still be useful for them to avoid significant waste of time when they encounter such paths. The detection of infeasible paths also plays an important role in other applications of program analysis [25, 26, 105, 178].

Current techniques in this area usually perform dataflow analysis to generate constraints from control flow graph and identify infeasible paths by verifying the unsatisfiability of the constraints [39, 87]. However, in general, detection of infeasible paths is theoretically unsolvable [79]. It is not possible to use constraint solvers or theorem provers to determine the infeasibility of all paths fully automatically.

1.1.3 Buffer Overflow Detection

Buffer overflow vulnerability has never been fully resolved in the past 20 years. The related cyber-attacks caused massive losses due to leaking critical information or losing users' trust in the long run. Many security associations rank buffer overflow at a *severe* level: Common Weakness Enumeration lists it among the top 25 most dangerous programming errors; Open Web Application Security Project (OWASP) listed it as the fifth most serious web application

vulnerability, and buffer overflow occupied more than 50% of the newly discovered vulnerabilities in the reports from SecuriTeam [72], US-CERT[19] and CVE [2]. Most buffer overflow vulnerabilities occurred in C/C++ programs. C/C++ grants users excellent flexibility and high runtime performance. Thus, many enterprises use C/C++ to develop applications. However, the trade-off is that C/C++ does not have sufficient build-in security mechanisms, such as buffer boundary checking, to ensure code security. Therefore, if attackers find any buffer without proper boundary protection, they can exploit the buffer and overrun the memory.

To detect buffer overflow, a lot of existing approaches uses static analysis to capture buffers without proper boundary protections. They have well studied the root causes for buffer overflows. Recent approaches have gradually found comprehensive solutions [89, 113, 115] at a theoretical level. However, their focus is more on the completeness while less on the practicality. The main drawbacks are their heavy overhead, and they may encounter unsolvable constraints [79]. The barriers make the existing approaches still far from a practical solution that could analyse large-scale programs automatically and effectively.

1.1.4 Buffer Overflow Removal

Besides buffer overflow detection, researchers have also explored other solutions to mitigate the threats of buffer overflow attacks [2]. Therefore, currently there are mainly three research directions:

- 1) Runtime time protection: Prevent buffer overflow attacks by creating a run-time environment, like a sandbox, so that taint input could not directly affect certain key memory locations;
- 2) Vulnerability detection: Detect possible buffer overflows and generates corresponding vulnerability reports;
- 3) Program transformation: Transform the original program by adding additional verification code or external annotations.

However, none of the existing methods fully resolved the threads from the buffer overflow attacks. For approaches on the first direction, as modern programs are becoming more complex, it is difficult to develop a universal run-time defence [2]. For approaches focusing on buffer overflow detection, even if buffer overflow vulnerabilities are detected, the vulnerable programs are still being used until new patches are released. For the third category, though it is well-motivated to add extra validation to guard critical variables and operations, the existing approaches will add considerable runtime overhead. For example, a recent novel approach that adds extra bounds checking for every pointer may increase the runtime overhead by 80% on average [19].

1.2 Objectives

The overall research objective of this thesis is to explore the feasibility of augmenting static analysis with empirical knowledge. More specifically:

Database verification: We would study the lifecycle of data in database applications empirically. Based on the discovered characteristics, one could detect database design errors effectively.

Infeasible path detection: We aim to create empirical code patterns of infeasible paths. Based on the patterns, it is possible to detect a large proportion of infeasible paths automatically, with refined and light-weighted static analysis techniques.

Buffer overflow detection: We would explore the characteristics of buffer overflow attacks and propose a set of patterns to cover the common buffer overflow vulnerabilities. So one can identify buffer overflow vulnerabilities more accurate yet scalable.

Buffer overflow removal: We would further investigate and abstract the characteristics of buffer overflow and propose an approach to remove the common buffer overflow vulnerabilities automatically.

1.3 Major Contributions

In this thesis, we propose four approaches augmented with empirical knowledge to serve important purposes in software engineering. These approaches have been published in international journals and conferences. The major contributions of this thesis are:

- 1) **A novel approach for database design verification:** We propose a domain-specific approach for database design verification. The approach comes from the need of providing transactions in any database application to support data lifecycle for all the data stored in its database. The approach encapsulates such data lifecycle into empirical patterns. It is effective to these patterns identify potential errors in the database design. Though the basis of the approach is straightforward, we proved its practicability through case studies. This approach can capture design errors like essential functionalities missing, transactions incompleteness or data redundancy. This work has been published in [192].
- 2) **A novel approach for infeasible paths detection:** From the large-scale empirical study of real systems' code, involving both program analysis and manual analysis, we find that the infeasibility of almost all program paths can empirically be determined using syntax analysis and a limited symbolic evaluation. We characterize infeasible paths in real systems based on two patterns that are detectable using limited symbolic

evaluation. Based on this characterization, it proposes a novel method called **Detection of Infeasible Paths Using Patterns with Limited Symbolic Evaluation (DIPLE)** to detect infeasible paths through recognizing the two patterns. The current characterization and method significantly enhances both the precision and coverage of our earlier proposal [136]. In the designed experiments, all infeasible paths detected by DIPLE are truly infeasible, and DIPLE detected all the infeasible paths from sets of paths selected randomly and according to commonly used path coverage criteria. The experiments also showed scalability and efficiency on the proposed method. This work is under the review of [184]

Additionally, we also extensively reviewed the related concepts, approaches and available tools for this topic. This work has been published in [190] and awarded as the best student research paper of the conference. An extended paper is published in [189].

- 3) **A novel approach for buffer overflow detection:** We make one step further to the area of buffer overflow detection, by proposing a method with more scalability while not sacrificing detection precision. The initiative is from an experimental study of those vulnerable codes. The study indicated most of the buffer overflow could be empirically characterized into three patterns that are detectable by syntax analysis and a restricted symbolic evaluation. Based on the findings, we propose a novel method to detect paths containing vulnerable buffer overflows through recognizing the patterns. The proposed method is scalable because it only evaluates selected paths and predicates. The proposed approach also offers high detection accuracy because (1) it performs a path-sensitive analysis and considers aliased pointers or references; (2) loops are abstracted to ensure high code coverage. The corresponding experiment proves the above statement. This work has been published in [191].
- 4) **A novel approach for buffer overflow removal:** The proposed approach first detects vulnerabilities in a given program. Defensive code is then patched to the appropriate program locations to remove the vulnerabilities. We have evaluated the proposed approach against both benchmark and industrial C/C++ applications. The results show that the proposed approach is effective: it can remove all the detected buffer overflow vulnerabilities in the studied subjects. This work has been submitted to [182].

1.4 Thesis Organization

This rest of the thesis is organised as follows. Chapter 2 provides the required background. It reviews theories and techniques of static analysis and design verification. Then, it discusses the domain-specific problems as (1) database design (2) infeasible paths issue and (3) buffer overflow vulnerability.

Chapter 3 starts with the need for supporting data lifecycle in a database application and the transaction patterns for providing it. Based on the proposed lifecycle patterns, we propose an approach for database design verification. The evaluation consists of two case studies and a comparative discussion.

Chapter 4 presents the limited symbolic evaluation based infeasible path patterns to characterize infeasible paths in real systems. Based on the patterns, we propose a method DIPLE to detect infeasible paths in a given set of paths from intra-procedural or inter-procedural control flow graphs. We conduct a large-scale experiment to demonstrate the efficiency of the proposed method.

Chapter 5 presents three code patterns and a restricted symbolic evaluation to identify patterns in real systems. Based on the patterns, we further present a method to detect buffer overflows in a given program. We conduct an experiment with both industrial and benchmark systems to show the proposed method enhance both the detection accuracy and scalability.

Chapter 6 further explores the buffer overflow issue and presents the rules for preventing the occurrence of buffer overflow attacks. Based on these rules, we present an approach that detects overflow vulnerabilities and removes detected vulnerability automatically. We conduct an experiment with both industrial and benchmark systems to show the effectiveness of the proposed method.

Chapter 7 reviews the related theories and compares our research findings presented in this thesis with related work. Three parts of related work are discussed: techniques of infeasible path detection, techniques of addressing buffer overflow vulnerability and techniques of enhancing the design quality of database applications.

Finally, Chapter 8 summarizes the contributions of this research and discusses the potential directions for future work.

The tables and figures that report the results of the BP0, CtP2 and Random experiments for Chapter 4 are shown in Appendix A to C.

All the notation conventions used throughout this thesis are listed in Appendix D.

Chapter 2

BACKGROUND

This chapter provides background information on static analysis and design verification. Section 2.1 reviews the fundamentals. Section 2.2 provides an overview of the research domains we are targeting in this thesis, including the issue of infeasible path, the vulnerability of buffer overflow and the design of database.

2.1 Program analysis

Program analysis is a set of essential techniques for software automation. Program analysis has two main categorizations as static program analysis [50, 70, 166] and dynamic program analysis [33, 43, 103].

Dynamic program analysis is the one that involves program executions with concrete inputs. So it can replace program's nondeterministic behaviours (e.g.: third party library, native function calls) with concrete values. This simplifies the analysis and makes the analysis more practical when applied to large industrial systems. However dynamic program analysis has its shortcomings. It generates runtime overhead and its precision limits to the selected concrete inputs.

On the contrast, static program analysis refers to techniques approximating program's runtime behaviour without executing the program. Static analysis is performed at compile-time. So it will not bring in runtime overhead. Well-modelled static analysis can characterize all executions, but it usually suffers from precision and scalability due to the complexity of the test programs.

These two kinds of analysis have their own strength, weakness and applicable areas. They are usually mutually supplementary, but in this thesis and below, the scope is on the static analysis part and we also only review the background for static program analysis.

2.1.1 Control flow analysis

Control flow analysis (CFA) [102, 128] is used to calculate the control flow of a program. The latter refers to the execution relationship between program statements, and it is usually expressed as a standard model of **control flow graph (CFG)**. Control flow analysis is fundamental of static program analysis. The essences include: (1) Identify the execution

sequence, for instance, execution path of statements, call sequence of functions and class hierarchy; (2) Help to comprehend complex program structures, for instance, loop structures; (3) Calculate control dependence, for instance, parallelization of threads, dominators and post-dominators.

A program can be modelled as a CFG, which is a directed graph. CFG is usually expressed as a four-tuple $G = \langle V, E, 'start', 'end' \rangle$, in which $V = \{v_i\}$ is a set of nodes, $E = \{e_i\}$ is a set of edges, 'start' is the unique graph entry node while 'end' is the unique exit node.

Nodes can have a one-to-one mapping relationship with program statements or a node may refer to a basic block which is a set of program statements without control statements (e.g.: *if*, *else*, *while*, *for*). Normal node (except predicate node) in a CFG is usually represented by a circle symbol "○" in a CFG. **Predicate nodes** fork the control flow after checking its predicate condition. Each out-going edge from predicate nodes is a **branch**. Predicate node is usually represented by a diamond symbol "◇" in a CFG.

Edge stands for the flow of control from one node to another. It usually uses a directed arrow line to represent one edge. A **path** $(x, v_1, v_2, v_3, \dots, y)$ that is a composition of edges represents one way to traverse the CFG from arbitrary node x to node y . A **completed path** is the one starts from the 'start' node and ends at the 'end' node. A completed path represents one of the execution traces. To count and extract paths from a CFG relies on path coverage criteria. Typical criteria include basis path criteria [145, 168], CtPx path criteria [32] and random path criteria [134].

The intra-procedure control flow analysis [155] analyses each procedure individually and isolated. It ignores the transfer of control due to functions calls/returns. There is a set of basic control dependency concepts. In a CFG, for arbitrary two nodes x and y , if x lies on every path from the 'begin' node to y , then x **dominates** y . Correspondingly if, y lies on every path from x to the 'end' node, then y **post-dominates** x . Additionally y is control dependent on x if and only if the following hold:

- 1) There is a path from x to y such that y post-dominates every node on this path.
- 2) The node y does not post-dominates x .

Informally, y is control dependent on x if there are two edges out of x and following one edge would reach y while following the other would not reach y [56]. There are some related concepts, including [1]:

- 1) Node x **strictly dominates** y if x dominates y and x does not equal y .
- 2) The **immediate dominator** of a node y is the closet dominator of y on any path from 'start' node to y .

- 3) A **dominator tree** is a tree in which this node immediately dominates all of its children nodes.
- 4) The **dominance frontier** of a node x is the set of all nodes y such that x dominates an immediate predecessor of y , but x does not strictly dominate y . It is the set of nodes where x 's dominance stops.

Figure 2-1 illustrates a procedure of Fibonacci calculation with its control flow graph. The program starts with an additional added entry node. The control flow forks at predicate nodes node2 and node7; the left out-going edge is the TRUE branch while the right side one is the False branch. The edge from node10 to node7 is a back edge, indicating a typical loop structure. Regarding the control dependency, the examples include: node11 is control dependent on node7; node7 is the dominator of node4, node5 and node6. With a CtPx path criteria and x equals to 1 [32], there are three completed paths in this CFG: $(start, 1, 2, 4, 5, 6, 7, 8, 9, 10, 7, 11, end)$, $(start, 1, 2, 4, 5, 6, 7, 11, end)$ and $(start, 1, 2, 3, end)$.

Figure 2-1. Sample program with its CFG

There are many novel algorithms to perform control flow analysis at the intra-procedural level [35, 55, 80, 154]. One popular approach is from Ferrante et al. [72]. Their contribution is providing a precise way to “reverse” CFG. In this “reversed” CFG, it has been proven it would be more efficient to calculate the control dependency [55, 145].

However, in order to function effectively on the entire program, it requires inter-procedural control dependences. The main objective here is to calculate the impact of functions calls and global attributes. Many definitions and approaches have been proposed [91, 122, 154, 155],

among which Sinha et al. [155] proposed two well-known approaches. The first approach connects the CFGs of called procedures to the CFG of the caller at each calling point. Thus, it expands a normal CFG into an inter-procedural inline flow graph (IIFG). This approach is context sensitive and offers high precision. However, this approach might be unordinary expensive if the program is complex. The second approach is a contrast to the first one. It ignores context-based distinctions and computes statement-based control dependencies so that it is a trade-off between analysis scalability and precision.

2.1.2 Data flow analysis

Computer program is like a pipeline that connects input and output and with data flowing through it. Data flow analysis [21, 77, 147] is the process to gather information about the impact of data to program's runtime behaviour. Data flow information contributes in activities such as testing and compiler optimization.

Data in a program may refer to (1) a single data item or (2) a combination of data items. Usually the first one is called a variable, and the second one is called an expression. Variables propagate along an execution path and normally would experience four scenarios:

- 1) Declaration: compiler reserves the memory space for a variable, and later this variable cannot be re-declared.
- 2) Definition: a value is assigned to the variable, and later this variable could be redefined.
- 3) Use: a program statement references a variable without modifying it.
- 4) Destruction: compiler frees the memory allocated for the value of a variable.

The major objective of data flow analysis is to find anomalies in data flow regarding the above four scenarios. Such anomalies may cause many problems during both compiling time and runtime, including:

- 1) Causing syntax errors during the compiling time, for instance, use or defining a variable before its declaration.
- 2) Downgrade runtime efficiency, for instance, a number of redundant definitions.
- 3) Causing logic problems, for instance, wrongly paired variable definition and use.
- 4) Causing unexpected runtime exceptions, for instance, use a variable after its destruction.

A number of models and approaches are available to describe the above problems formally. The basic models include live/dead variables analysis, available expressions analysis, reaching definition analysis, point-to and alias analysis [104]. To perform data flow analysis over these models, a general analysis framework will first setup data flow equations for each node on the

CFG; then solve these equations by iteratively propagating CFG and calculating the impact of traversed nodes until the analysis reaches a termination condition [106]. There are a number of novel approaches inherited from this framework. Pure intra-procedural data flow analysis assumes input parameters, global attributes and return values of functions calls can all be treated as free variables; it also assumes there is no side-effect when calling an arbitrary function.

However when facing industrial programs, it usually requires feasible approaches to perform data flow analysis at inter-procedural level. The basic models and concepts are same with intra-procedural analysis, but the challenge is how to dealing with the flow of data across procedure boundaries. One novel approach is from Allen et al. [20]. They organised the invocation sequences in reverse order so that the data flow information of all the invoked functions would be ready before processing the caller functions. Another approach is from Myers [131]. He extended normal CFGs into an inter-procedural inline flow graph so all the invoked functions would be substituted inline at the calling point. However, the two approaches are not scalable enough in real cases because they are trying to organise a super-graph for the entire program. When the program is complex, the size of the super-graph will increase tremendously fast. Huang [96] shows how to detect dataflow anomalies, during which several important techniques for inter-procedural analysis are introduced, including inter-procedural alias analysis, whole program control flow construction and program segmentation.

One alternative way is to perform demand-driven inter-procedural data analysis to reduce the computational cost [65, 66, 95, 99, 147]. For example, Reps [147] showed that a large subset of data flow analysis problems, such as truly-live variables [106] and copy constant propagation [148], are inter-procedural, finite, and distributive. These problems can be solved precisely in polynomial time after being transformed into a graph reachability problem.

Another alternative way is to involve manual work. One attempt is to equip the inter-procedural data flow analysis with mock objects [4, 141]. Programmers create mock objects to simulate the runtime behaviour of real objects when the code of the real objects is complex. A recent example is from Kunal et, al. [158]. They proposed to augment the data flow analysis with mock objects to generate test generation automatically for database applications.

2.2 Domain background

2.2.1 Database design verification in relational model

Database design [3, 83, 160] is to specify the detailed data model of data repository. There are many types of database model. Relational model is one of the most matured types and has been in widespread use for more than three decades. Design a database in a relational model usually

means to specify a structure in tables and views. Designers should determine the relationship between different data elements and specify the logic to fill data into tables and views. During this process, designers should verify the design to avoid a number of pitfalls, such as:

- 1) Vulnerable operations that violate database integrity: entity integrity, domain integrity and referential integrity;
- 2) Operations that generate redundant data eventually or operations that may cause data loss or other security issues;
- 3) Poorly designed SQL (structured query language) statements or database structure, which downgrades the database performance;

If the system is complex, database design verification could be difficult and requires additional tools for assistant [48, 49, 53, 74, 93, 94, 165]. However, recent reports [139] shows that there is a false impression that the database verification is the responsibility of the software engineer/designer. In practice, the verification is only carried during the unit testing stage when the most programming work has been nearly completed. The importance of early stage checking with more involvement of domain experts and even end-users are usually ignored. Database design specification is the core work of the early stage database design. One of the current research trends [139] is to explore the verification of database design specification, and therefore, to eliminate the possible design anomalies at early stage. A more detailed literature review is covered in Chapter 7.3.

2.2.2 Infeasible paths

An infeasible path is a path in the CFG of a program that cannot be exercised by any input values. Figure 2-2 shows an infeasible path $p = (entry, 1, 2, 3, 4, 5, 6, exit)$ in a CFG. This is because we cannot find any input x satisfying $x \geq 0$ and $x < 0$ jointly.

The existence of infeasible paths has a significant impact to many software engineering activities. If there is an approach to detect infeasible paths effectively during the process of optimization, the corresponding program can certainly be further optimised. In software testing, the structural test coverage can be much accurately computed if infeasible paths can be detected more accurately. In the automated generation of structured test cases, much time can be saved if more infeasible paths can be detected. It can also help programmers for better code comprehension as it can identify spurious paths inserted due to obfuscation codes. In software verification, detecting and eliminating infeasible paths will help to enhance the verification precision and speed. There are many more areas like security analysis [143], web application verification [118-121], database application design [138] that can be helped by the detection of infeasible paths.

To detect infeasible paths in real programs, manual checking is not enough. The infeasibility may be caused by complex data structures and dependency. Consider the example in Figure 4-3, the program shows a customized procedure, *process*, that calls a reusable procedure, *analyseB*, in a library to decode *buffer*. There is a path $\pi = (a_4, a_5, \text{entry_analyseB}, b_1, b_2, b_3, b_4, b_6, b_7, b_9)$ is infeasible path. But it is difficult to detect such infeasible path because this program is inter-procedural and it also contains a complex loop.

To detect infeasible paths, there requires considerable effort to present them formally in symbolic expressions or constraints for further verification by heuristics, predefined rules or even standard theorem provers. If the verification returns negative results (e.g.: “Invalid” answer from theorem provers), the path is then considered as infeasible. Such verification model is undecidable in general (A more detailed literature review is Chapter 7.1). However, it is still possible to have practical approaches that are not theoretically complete to detect infeasible paths. Later in Chapter 4, we will propose an approach of this kind.

Figure 2-2. An infeasible path

2.2.3 Buffer overflow vulnerability

Buffer overflow occurs any time when a program fails to prevent input from exceeding intended buffer size(s) and accessing critical memory locations. The attack usually starts with the following attempts [19, 163]:

- 1) Exploit a memory location in the code segment that stores operations accessing memory without proper boundary protection. For example, a piece of code allows writing arbitrary length of user input to memory.

- 2) Find a desired memory location in data segment that stores (a) an important local variable or (b) an address that is about to be loaded into the CPU's Extended Instruction Counter (EIP register), an address storing the connection information of a socket for instance.

Attackers attempt to calculate the distance between the above two memory locations. Once such locations and distance are discovered, attackers construct a piece of data of length $(x1+x2)$. The first $x1$ bytes of data can be any characters and is used to fill in the gap between the exploited location and the desired location. The second $x2$ bytes of data is the attacking code which could be (a) an operation overwriting a local variable, (b) a piece of shell script hijacking the system or (c) a handle redirecting to a malicious procedure.

In addition, regarding the target application architecture, operating system and memory region, attackers require various other techniques to complete a buffer overflow based attack successfully. For example, the attacks over stack memory and heap memory are different [171]; a piece of code safe on Linux may be vulnerable on Windows. However, no matter how complex the techniques are, a successful buffer overflow attack requires exploiting vulnerable code first, more specifically, finding an execution path containing vulnerable buffer writing. Then attackers manage to send the malicious code along the exploited path to overrun the buffer. Therefore, to prevent buffer overflow attack, it is necessary to ensure buffer writing operations are accessible only after proper validations.

Figure 2-3 shows an example of one buffer overflow vulnerability. For the sample procedure `_encode`, node `s1` uses the C string function `strcpy` to copy a block of data to a destination buffer. Such a function call is vulnerable because `strcpy` does not have any built-in mechanism that prevents over-sized data from being written to the destination buffer. Additionally, before node `s1` there is no explicit validation to constraint the relationship between the `pvpbuf`'s size and `req_bytes`'s length. At node14, the variable `req_bytes` is defined as an array of length 1024 and it receives a bulk of data size of 1024 from user input via the command-line. Along the path `(n9, n10, n11, n12, n13, n14, n15, n16, s1)`, the variable `pvpbuf` is defined as an array of size `c2`. However, there is no validation to ensure `c2` is larger than 1024, in order to prevent `pvpbuf` being overwritten. Therefore if any execution path that contains the subpart `(n9, n10, n11, n12, n13, n14, n15, n16, s1)` is discovered, the system running this procedure `_encode` becomes vulnerable.

Figure 2-4 further illustrates this example. It demonstrates the stack frame for `_encode` in a common x86&windows architecture. The stack grows down from high address to low address, where storing the function parameters, the local variables and the `ret--EBP` address of the caller. Based on this stack frame, an x86 CPU would move to the caller's EBP address for the

next instruction after executing `_encode`. If an attacker finds that, at node *s1*, the local buffer *req_bytes* lacks protection; he/she would exploit and overrun this buffer with malicious data. The malicious data will smash the stack and overwrite the *ret* with a new memory address. This address points to a malicious program, which usually is a backdoor application. Later after returning from `_encode`, the system would be misled to execute that backdoor application. Therefore, the attack successfully hijacks the system. For more information, including the approaches to enhance the attack efficiency, interested readers may refer to other documents [140, 163].

It is also discovered by the existing work [113, 117] that there are cases when buffers may be overwritten by constant variables. This means such overflows are input independent. Though these cases will cause run-time errors, they are out of attackers' control. Therefore, we will not consider them here.

A more detailed review of related work for dealing with buffer overflows is covered in Chapter 7.2.

```

#define min_size 10;
#define max_size 1024;
/*
  addr is an existing array with the size of ADDRSIZE,
  n2 and n16 are two basic blocks that are not related to
  pvpbuf and req_bytes
*/

int _encode (bool bslashmode, char *addr,
             int ADDRSIZE){
  /*n1*/ do{
  /*n2*/   BLOCK1;
  /*n3*/ } while(...);
  /*n4*/ int c1=0; int c2=0;
  /*n5*/ int PSBUFSIZE= max_size;
  /*n6*/ int *pvpbuf;

  /*n7*/ if (bslashmode == TRUE){
  /*n8*/   pvpbuf = new int [PSBUFSIZE];
  }else{
  /*n9*/   while(c1< ADDRSIZE){
  /*n10*/     if(addr[c1]!='#')
  /*n11*/       c2++;
  /*n12*/     c1++;
  }
  /*n13*/   pvpbuf = new int [c2];
  }

  /*n14*/ char req_bytes[1024];
  /*n15*/ scanf("%1024s", req_bytes);
  /*n16*/ BLOCK2;

  /*s1*/ strcpy(pvpbuf, req_bytes);
  }//END encode

```

Figure 2-3. A sample code with exploited buffer overflow vulnerability

Figure 2-4. Memory arrangement of the sample code

2.3 Empirical approaches for Pattern Recognition

Static analysis and requirement analysis are among the fundamental aspects of program analysis. They provide means to verify software status against desired standards or constraints. They are the necessary elements to guarantee software quality.

However these analysis methods tend to be general and formal. The real situations are far more complex. Different software is built upon different domain information. It is essential to extract such domain information and conclude them as empirical knowledge, which is the data from observation or experimentation, to justify a belief in the truth or falsity of an empirical claim [149]. In many other areas, like healthcare or physics, the empirical knowledge is widely adopted with formal diagnose/analysis procedure to improve the accuracy of the final result [135].

In program analysis, there are several important areas like infeasible paths detection, vulnerability detections, where traditional formal approaches cannot resolve the problems perfectly. Therefore, we need to study and conclude the empirical knowledge within these areas and use them to improve the state-of-the-art.

We adopt a widely used guideline [108] for conducting empirical studies in software engineering which hopefully will improve the quality and quantity of empirical studies. This guideline mainly consists of the following steps:

- 1) **Experimental context:** One needs to record the information that is about the industrial context where the experiment is conducted.
- 2) **Experimental design:** One should record the population under analysis and the sampling technique used. It is also recommended to record down what intervention and methods have been used to reduce bias to determine the sample size.
- 3) **Conduct of the experiment and data collection:** One needs to write down the measures for the outcome of a study.

- 4) **Analysis:** Procedures used to conduct and control for multiple testing should be specified.
- 5) **Presentation of results:** A reference should be provided to show all statistics. The magnitude of effects, threats to validity [47], and approach limitations should also be represented.
- 6) **Interpretation of the results:** One need to analyse the results and show the significance and also the limitations of the proposed approaches against those existing works in literatures.

There are various ways of carrying out studies to get empirical knowledge. But the most common ones are [135]:

- **Experiments:** the purpose of an experiment is to make observations that are otherwise hard to obtain.
- **Survey:** a survey is a retrospective study of a situation that investigates relationships and outcomes. It is helpful to obtain the current status in the areas.
- **Case studies:** A case study is an observational study that investigates a contemporary phenomenon within a real-life context to support a more general assumption or conclusion for that context.

In this thesis, we use the above guideline and methods to explore empirical patterns in database design verification and static code analysis. The patterns describe important properties in database design verification and static code analysis. They are modelled from some initial observations, and later are iteratively refined to fit the context better. Combining with traditional formal program analysis tools and methods, we are going to propose new approaches that can improve state-of-the-art in several key areas in software engineering: database design verification, infeasible path detection and buffer overflow handling.

Chapter 3

DESIGN VERIFICATION FOR DATABASE APPLICATION THROUGH ANALYSING THEIR SUPPORT TO DATA LIFECYCLE

In this chapter, we proposed a domain specific design verification approach for the development of database applications. The approach is based on data life cycle patterns in database applications. Through experiments, we find the proposed method is useful in detecting missing essential functionalities and design errors.

3.1 Data Lifecycle and Transaction Patterns for Supporting

A database application maintains its data in a database. The data is defined in a database schema. In general, a database schema defines the data in terms of record types each of which comprises of a set of attributes. All the attributes and record type defined in the schema represent essential data to the application. None of these attributes and record types should be redundant. Data maintain in a database is usually dynamic and changeable. These data can also become obsolete. Once a piece of data becomes obsolete, it should be removed from the database so that the database will not grow unlimitedly. These characteristics imply that both attribute and record type have a lifecycle that starts from being inserted, via being used and optionally being updated in any number of times in and any sequence, to being deleted. Such a lifecycle is called lifecycle of attribute and record type or more generally called data lifecycle.

More formally, the lifecycle of attribute and record type can be modelled using the state transition diagram shown in Figure 3-1. The events indicated in the diagram are as follows:

- 1) Insert: For lifecycle of attribute, an occurrence of insert event happens when the value of the attribute is first defined. For lifecycle of record type, an insert event happens when a new record of the record type is inserted.
- 2) Update: For lifecycle of attribute and record type, the occurrence of update event happens when a value of the attribute and a record of the record type respectively are updated.

- 3) Use: For lifecycle of attribute and record type, an occurrence of use event happens when a value of the attribute and a record of the record type respectively influence one of the following:
 - a) An insertion, deletion or updating of other attribute or record type respectively;
 - b) A value of variable that is outputted to the external environment.
- 4) Delete: For lifecycle of attribute, an occurrence of delete event happens when a record of the record type that possesses the attribute is deleted. For lifecycle of record type, an occurrence of delete event happens when a record of the record type is deleted.

Except the event update and the stage updated, all other events and stages in the diagram are mandatory.

Figure 3-1. The lifecycle of attribute and record type

The lifecycle of attribute and record type implies directly that any database application must provide transactions for the following:

- 1) There are insert operations on each attribute and record type
- 2) Value of each attribute and record type would be retrieved for other insert, update and delete operations.
- 3) There are delete operations for each record type.

It also implies that though it is not necessary, any database application usually should also provide transactions update on each attribute and record type. We formalize these provisions as the following four transaction patterns that could be widely adopted in most database applications.

Transaction Pattern 1 – Provision for Insertion. In a database application, for each attribute and record type, there must be a transaction that performs insertion on the attribute and the record type respectively.

Transaction Pattern 1 is to provide the functionality for inserting a new attribute value or a new record. This functionality is essential as otherwise its value of an attribute or a record type is left undefined.

Transaction Pattern 2 – Provision for Use. In a database application, for each attribute and record type, there must be a transaction that performs retrieval on the attribute and record type respectively such that the information retrieved influences one of the following:

- 1) Insertion, updating or deletion on other attributes or record types respectively.
- 2) Information outputted to the external environment.

Transaction Pattern 2 is to provide the functionality for using an attribute or a record type. This functionality is essential as otherwise the attribute or the record type is redundant. Note that only external use is considered and use for itself is not considered.

Transaction Pattern 3 – Provision for Deletion. In a database application, for each record type, there must be a transaction that performs deletion on the record type.

Transaction Pattern 3 is to provide the functionality for removing obsolete records. Though a record can be marked as logically inactive by setting the value of one of its attributes that serves as a flag instead of physically deleted, these logical inactive records must still be deleted after some time. Otherwise the database may grow unexpectedly. Hence, removing obsolete records from a database is still an essential functionality.

Transaction Pattern 4 – Likely Provision for Updating. In a database application, for each attribute and record type, usually there is a transaction that updates on the attribute and the record type respectively performed.

Records or attributes captured in some transactions might not be modifiable. For example, when receiving an order to buy company A's stock, usually stock exchange systems will lock related data in the database. Nevertheless, other record types and attributes should still be modifiable. Transaction Pattern 4 is to provide such likely requirement.

3.2 Design Verification via the Transaction Patterns for Providing Data Lifecycle

A transaction can be identified from the use-cases in a system. A step in a use-case that starts from a request sent by a main actor to a system and ends with the system responds to the actor is a transaction [42, 62, 139]. The activities included in the step are:

- 1) the main actor sends requests and data to the system,
- 2) the system validates the request and data,

- 3) the system performs internal state change and update databases,
- 4) the system responds to the actor with results requested.

The identification can be carried out manually or automatically [139].

We propose an approach for verifying the design of a database application using the transaction patterns for data lifecycle discussed in the last section. Once the design of a database application is completed, for each attribute and record type in its schema, through the above-mentioned identification of transactions from use-cases designed, we verify the existence of transactions following each pattern as follows:

- 1) Transaction Pattern 1: The existence of a transaction that performs insertion on the attribute and the record type respectively, in a database application.
- 2) Transaction Pattern 2: The existence of a transaction that performs retrieval on the attribute and the record type respectively such that the information retrieved influences one of the following cases, in a database application:
 - a) Insertion, updating or deletion on other attribute or record type.
 - b) Information outputted to the external environment.
- 3) Transaction Pattern 3: The existence of a transaction that performs deletion on the record type, in a database application.
- 4) Transaction Pattern 4: A transaction that performs updating on the attribute and the record type respectively, in a database application.

Clearly, attribute represents the lowest-level of data in a data application. Record type represents higher-level of data that comprises of lower-level data represented by attributes. Hence, any missing of transaction for supporting data lifecycle at record type level is more serious than attribute level.

The missing of transactions following Transaction Pattern 1, 2, 3 and 4 at attribute and record type level implies the missing of insertion, use, deletion and updating functionalities respectively. Different combinations of verification results have different implications. We classify them according to their level of severity as follows:

- 1) Undefined Data: If there are transactions for an attribute or a record type that follow Transaction Pattern 2, 3 or 4, but there is no transaction for it that follows Transaction Pattern 1, then it implies that the data represented not even defined but are used, deleted, or updated. This is the most severe error. The seven combinations of verification results fall under this error are shown the first seven in Table 3-1 for the transaction pattern

concerned, in which “T” and “F” mean “exists” (true) and “does not exist” (false) respectively.

Table 3-1. Type of errors

S/N	Transaction Pattern 1	Transaction Pattern 2	Transaction Pattern 3	Transaction Pattern 4	Type of Errors	Implication
1.	F	T	T	T	Undefined data	Data can be updated, used and deleted, but cannot be inserted.
2.	F	T	F	T	Undefined data	Data can be used and updated, but cannot be inserted and deleted.
3.	F	T	T	F	Undefined data	Data can be used and deleted, but cannot be inserted and updated.
4.	F	F	T	T	Undefined data	Data can be updated and deleted, but cannot be inserted and used.
5.	F	T	F	F	Undefined data	Data can be used, but cannot be inserted , updated and deleted.
6.	F	F	F	T	Undefined data	Data can be updated, but cannot be inserted , updated and deleted.
7.	F	F	T	F	Undefined data	Data can be deleted, but cannot be inserted , used and updated.
8.	T	F	T	T	Data redundancy	Data can be inserted, updated and deleted but cannot be used.
9.	T	F	T	F	Data redundancy	Data can be inserted and deleted but cannot be used and updated.
10.	T	F	F	T	Data redundancy	Data can be inserted and updated but cannot be used and deleted.
11.	T	F	F	F	Data redundancy	Data can be inserted, but cannot be used , updated and deleted.
12.	T	T	F	T	Non-removable data	Data can be inserted, used and updated, but cannot not deleted.
13.	T	T	F	F	Non-removable data	Data can be inserted and used, but cannot be deleted and updated.
14.	F	F	F	F	Data design redundancy	Data specified in the schema but not inserted, used, updated and deleted.
15.	T	T	T	F	Non-modifiable data	Data inserted, used and updated, but cannot be updated.

- 2) **Data Redundancy:** If there are transactions for an attribute or a record type that follow Transaction Pattern 1, but there is no transaction for it that follows Transaction Pattern 2, then it implies that the data represented is defined but not used. This implies that the data is redundant. The four combinations of verification results that fall under this error are shown after the undefined data error in Table 3-1.
- 3) **Non-removable Data:** If there are transactions for an attribute or a record type that follow Transaction Pattern 1 and 2, but there is no transaction for it that follows Transaction Pattern 3, then it implies that the data represented is defined and used but cannot be deleted. This implies that the data is non-removable. The two combinations of verification results that fall under this error are shown after the redundancy data error in Table 3-1.
- 4) **Data Design Redundancy:** If there is no transaction for an attribute or a record type that follows any of the transaction patterns, then it implies that the data is designed in the schema but not implemented at all. As shown in Table 3-1, there is only one combination of the verification result that falls under this error.
- 5) **Non-modifiable Data:** For the same data, if there are transactions that follow Transaction Pattern 1, 2 and 3 while there is no transaction follows Transaction Pattern 4, then it implies that the data represented is defined used and removable but cannot be modified, which implies the data is non-modifiable. As shown in Table 3-1, there is only one combination of the verification result that falls under this case. As some data cannot be amended once it is captured, this case is considered as a warning error.

Based on the verification results, designers can investigate and examine the design to find out the causes as follows:

- 1) As all transactions are provided in the use-cases. For error type 1, 2 and 3, usually, the design and user requirements are investigated to find out whether it is due to the missing of use cases or transactions in use-cases already provided. If so, the missing use-cases and transactions will then be added to rectify the design accordingly. For some cases, it could be due to the redundant attribute or record type introduced in the schema. If so, it will also be removed and rectified accordingly.
- 2) For error type 4, usually, it is due to redundant attribute or record type provided. If so, it will also be removed and rectified accordingly. For rare cases, it is could be due to the missing of use-cases or transactions. If so, these use-cases or transactions will also be introduced accordingly.

- 3) For error type 5, it is only a warning error. The missing of transactions could be intentionally due to the nature of data. Otherwise, it will also be rectified accordingly in a similar way as error type 1, 2 and 3.

The proposed method is based on fundamentally characteristics, and therefore can be applied across database applications. It can be fully automated. We have implemented a prototype system to implement the proposed approach. The input to the prototype system is the informal use case descriptions written upon the Use Case Template specified by [139]. Such template mechanism provides formal means to model business transactions within use case specifications. Figure 3-2 (a) shows such template and Figure 3-2 (b) shows an example of using such template to specify a use case: adding student to a class. Such input covers all the use cases in a system and a database schema for the system. It is stored in text files, named *transaction descriptions*. Adopted the work in [139] and followed its usage guidelines, for each transaction, our system processes the corresponding *transaction descriptions* to automatically check the provision of insertion, updating and deletion in the transactions by scanning their textual specifications to detect following common words used for such purposes:

- 1) Insertion: “add”, “insert” and “create”.
- 2) Updating: “update”, “modify” and “change”
- 3) Deletion: “delete”. “terminate” and “remove”

The attributes and records operated on by these operations are identified by their names.

<p>MAIN SUCCESS SCENARIO</p> <ol style="list-style-type: none"> 1. step*. 2. TRANSACTION. TRANSACTIONAL RESOURCES (TRs): <i>list of TRs</i>. <ol style="list-style-type: none"> 2.1 step* (<i>possibly accessing TRs</i>). 2.2 BEGIN CONFLICT PREVENTION STAGE 2.3 step* (<i>possibly accessing, changing TRs</i>). 2.4 COMMIT. 3. step*. <p>EXTENSIONS</p> <p>(2.1-2.3) Primary actor indicates that he/she wishes to abort the transaction:</p> <ul style="list-style-type: none"> – step*. – TRANSACTION ABORT. – step*. <p>(2.1-2.4) System detects that TRs were changed by another actor:</p> <ul style="list-style-type: none"> – step* (<i>conflict resolution</i>). – TRANSACTION ABORT (<i>optional</i>). – step* (<i>optionally resuming at 2.x in an attempt to retry</i>). <p>(2.2) Exclusive access to TRs could not be obtained:</p> <ul style="list-style-type: none"> – step* (<i>conflict resolution</i>). – TRANSACTION ABORT (<i>optional</i>). – step*. <p>(2.3) Response time out:</p> <ul style="list-style-type: none"> – step*. – TRANSACTION ABORT. – step*. <p>(2.4) Failure persisting data:</p> <ul style="list-style-type: none"> – step* (<i>recovery</i>). – TRANSACTION ABORT (<i>optional</i>). – step* (<i>optionally resuming at 2.4 in an attempt to retry</i>). 	<p>MAIN SUCCESS SCENARIO</p> <ol style="list-style-type: none"> 1. (Trigger) Primary actor indicates that he/she wishes to add a <u>Student</u> from the <u>Student Registry</u> to a <u>Class List</u> 2. TRANSACTION TRANSACTIONAL RESOURCES: <u>Student Registry</u>, all <u>Class Lists</u>. <ol style="list-style-type: none"> 2.1 At the System's prompt, the Primary actor identifies a <u>Class List</u> and the student to be added (who we will call <Alice>); System ensures that the provided information is valid. 2.2 BEGIN CONFLICT PREVENTION STAGE 2.3 System adds <Alice> to the <u>Class List</u>. 2.4 COMMIT. 3. System notifies primary actor that <Alice> has been successfully added. <p>EXTENSIONS</p> <p>(2.1-2.3) Primary actor indicates to abort the transaction:</p> <ul style="list-style-type: none"> – TRANSACTION ABORT. <p>(2.1-2.4) System detects that TRs were changed by another actor:</p> <ul style="list-style-type: none"> – System notifies primary actor that TRs have been changed by another transaction. <i>Use case resumes at 2.1.</i> <p>(2.1)a The given id for <Alice> is not in the <u>Student Registry</u>:</p> <ul style="list-style-type: none"> – System informs primary actor that information of <Alice> is invalid—e.g. invalid student id. <i>Use case resumes at 2.1.</i> <p>(2.1)b <Alice> is already in the <u>Class List</u>:</p> <ul style="list-style-type: none"> – System informs primary actor that <Alice> is already in the <u>Class List</u>. <i>Use case resumes at 2.1.</i> <p>(2.2) Exclusive access to TRs could not be obtained:</p> <ul style="list-style-type: none"> – System notifies primary actor that the conflict prevention stage could not be entered. System asks the primary actor if he/she wishes the system to retry entering the conflict prevention stage. Primary actor acknowledges. [Extensions to this step are not shown.] <i>Use case resumes at 2.2.</i> <p>(2.4) Failure persisting data:</p> <ul style="list-style-type: none"> – System notifies primary actor that the transaction did not complete because the changes could not be persisted. – TRANSACTION ABORT.
---	--

(a) Use Case Template for Transaction Description

(b) An Example: add a student to a class

Figure 3-2. Use case template and an example [139]

3.3 Empirical Evaluation

To evaluate the feasibility of using the proposed approach and the usefulness of the proposed approach, we have conducted two case studies each of which uses the proposed approach to develop one database application. In the first case study, a student software project team augments the Unified Software Development Process [12, 97] (the Object-Oriented software development methodology that uses Unified Modelling Language) with the proposed design verification to develop a university library system as a requirement for their MSc program. The verification was carried out automatically. In the second case study, an industrial software project team augments the proposed design verification in their software development process to develop a real car rental system.

Table 3-2 shows the quantitative results of the two case studies. In the first case study, the student software project team verified the design of use-cases automatically using the proposed design verification. All the use-cases were specified informally as stated in [139]. The verification was carried out automatically using the prototype system described in the previous section that was developed by us. The proposed verification led to the discovery of the following use-cases that were missing in the original design:

- 1) Registration of a new book: The proposed verification found that the system has undefined data error on book record, that is, the record type book violates Transaction Pattern 1 but satisfies Transaction Pattern 2. After a further investigation, they found that the use-case for inserting data on a new book purchased was missing.
- 2) Extension of loan: The proposed verification found that the system has warning non-modifiable error on the due date of loan record. Further investigation found that the due date in the loan record should not be changed, but if a borrower requests for a change to a later date, it should be changed under a new use-case, called an extension of the loan that captures the new date separately. This led to the discovery of this use case.
- 3) Update of member particulars: The proposed verification found that the system has warning non-modifiable error on member record and its attributes. Further investigation found that except the member identification, all other attributes should be modifiable. This led to the discovery of this use case for the modification.
- 4) Removal of obsolete records: The proposed verification found that the system has non-removable error on all records and attributes. Further investigation found that this use-case should be provided to remove all obsolete records after a while.

In addition, as shown in Table 3-2, the proposed verification also helped to detect missing and redundant attributes and records. It also helped to correct some erroneous database operations specified in the original use-cases (the last two rows shown in Table 3-2).

Table 3-2. Statistics obtained from our case studies

	Case Study 1	Case Study 2
No. of missing use-cases detected	4	2
No. of missing record types detected	5	3
No. of missing attributes detected	12	8
No. of redundant record types detected	3	1
No. of redundant attributes detected	14	7
No. of missing database operations detected in use-cases	3	1
No. of redundant database operations detected in use-cases	1	0

In the second case study, an industrial software project team used the proposed method to verify the design of a car rental system. Their verification is to eliminate design errors in the high-level pseudo codes before they proceeded to the programming stage. The proposed verification helped to discover the following use-cases that were missing in the original design:

- 1) Cancellation of Rental: The proposed verification found that the system has undefined data error on the status in the rental record. Further investigation found that the status is to indicate whether a rental record is “active” or “cancelled”. If a car rental is cancelled for some reasons or due to data entry mistake, this attribute should be updated to “cancelled”. This led to the discovery of this use case.

- 2) Extension of Rental: The proposed verification found that the system has warned non-modifiable error on the date of return in the rental record. Further investigation found that the date of return should not be changed, but if a customer requests for a change of date of return to a later date, it should be extended in a new use-case, called extension of rental, that captures the new date of return separately. This led to the discovery of this use case.

Similarly, to the first case study, as shown in Table 3-2, the proposed verification also helped to detect missing and redundant attributes and records. It also helped to correct some erroneous database operations specified in the original use-cases.

In summary, the case studies found that the proposed design verification is straightforward and easy to carry out. It is very useful for designers to detect essential functionalities. These functionalities might be missed in the original design especially when they are not within the part of day-to-day functions. The proposed approach also helped to improve the accuracy of database and use-case design.

Our approach helps to detect design errors in the design stage. Hence, it helps to detect design errors in the earliest stage to ensure that such errors will not be carried forward to the later stages. This will improve the effectiveness of software design as it is well-known that detecting design errors earlier in the software development is the most cost effective way.

3.4 Conclusion

In this chapter, the main contributions include:

- 1) We have proposed a straightforward and yet useful method for verifying design of database applications at the earliest stage.
- 2) To the best of our knowledge, no such approach has been proposed. In opposing to the current approaches, based on the fundamental characteristics of database applications, the proposed approach can detect missing and erroneous functionality from the design automatically at the earliest stage in software development.

Design verification is extremely essential for the effectiveness and productive development of software systems. Further research includes the use of patterns for the verifications of other aspects in the design of database applications. We may also look into design verification for other domains.

Chapter 4

DETECTION OF INFEASIBLE PATHS USING PATTERNS WITH LIMITED SYMBOLIC EVALUATION

In general, the detection of feasibility of paths is theoretically unsolvable [79]. It is not possible to use constraint solvers to determine the infeasibility of all paths fully automatically. Current techniques are mostly proposed within the solution to larger problems (e.g., for code optimization purpose) in which detection of infeasible paths is useful. They are either based on data flow analysis or constraint propagation [39, 87].

We extend the rules for detecting infeasible paths employed in [39], that are also used as path-specific/ad-hoc pre-processing techniques for the same purpose in [151, 162, 173], and generalise them into two patterns for characterizing infeasible paths in real system code. The two patterns are detectable using only limited symbolic evaluation. Based on this characterization, we propose a novel method, called **Detection of Infeasible Paths Using Patterns with Limited Symbolic Evaluation (DIPLE)**, to detect infeasible paths through efficient recognition of the two patterns. The method proposed in this chapter significantly enhances both the precision and coverage of our earlier proposal [136].

To the best of our knowledge, we have not seen any approach that has been empirically shown to detect most of the infeasible paths in industrial software. From our large-scale empirical study of multiple and diverse systems, involving both program analysis and manual analysis, we find that, DIPLE shows high precision and scalability.

4.1 Infeasible Paths Characterization

We shall adopt the basic definitions for control flow graphs (CFG) from Sinha et al. [155]. An **infeasible path** is a path in a CFG that cannot be exercised by any input values. Next, we shall introduce our terminology before proceeding further.

In earlier work [136], an approach was proposed to detect sizable numbers of infeasible paths in an efficient manner based on common path characteristics. Some recent test case generators have also taken advantage of such common characteristics among infeasible paths to optimise the process of test case generation [151, 162, 173]. Our characterization aims at a more comprehensive coverage of such infeasible paths characteristics.

If we exclude programs that are intentionally and knowingly coded to contain infeasible paths (for example, infeasible paths introduced for control flow obfuscation [124]), from our observation, infeasible paths in real code can be classified into the following two patterns:

Pattern 1 - Flag-Based Inhibitor Pattern: The condition of a branch in a path is defined through referencing to a variable whose value is a constant conflicting with the branch condition.

Pattern 2 - Conflicting Branch Pattern: The conditions of some dependent branches in a path can never be satisfied jointly due to a conflict that can be evaluated using a linear constraint solver.

Before proceeding to a formal description of these two patterns, we shall define some terminologies.

A variable in a program is called an **input variable** if it is not defined in the program solely from constants and variables. Let v be an input variable defined at a node x in a CFG. A node y in the CFG **transitively references** to v if there is a path from x to y , a sequence of nodes, $y_0 = x, y_1, \dots, y_n = y$ in the path and a sequence of variables $v_0 = v, v_1, \dots, v_n$, such that $n \geq 1$, for each $j, 1 \leq j \leq n$, y_j defines variable v_j and references to variable v_{j-1} , and the sub-path from y_{j-1} to y_j is a definition-clear path with respect to v_{j-1} .

A variable v in a procedure is called a **flag** if all the nodes that define v do not transitively reference to any input variable. A predicate node is called a **flag-based predicate node** if it transitively references to only flags.

Figure 4-1a shows a procedure from [92] rewritten in Java. In its CFG shown in Figure 4-1b, both nodes 10 and 12 are flag-based predicate nodes. They only reference the flag *match* that is only defined by nodes that do not transitively reference to any input variable.

Constant propagation is commonly used in compiler for optimization purpose [169]. In this chapter, we use constant propagation in a much more restricted form. Our constant propagation refers to the process of evaluating the predicate value of a branch in a path, through backward propagation if the value is a constant. We call this restricted form of constant propagation, **limited constant propagation (l-constant propagation)**.

Figure 4-1. Flag-based inhibitor pattern

In Figure 4-1b, in path $\pi = (\text{entry}, 1, 3, 4, 5, 6, 7, 8, 10, 12, 13, \text{exit})$, the branch predicate of the branch at node 12 in π is $Q \equiv (\text{match} == 1)$. Applying l -constant propagation, we can evaluate that $Q \equiv \text{false}$ as follows. Propagating Q backward from node 12, the following steps are performed:

- 1) At assignment node 7, match is substituted with $(\text{match} + 1)$. Hence, $Q \equiv (\text{match} + 1 == 1)$.
- 2) At assignment node 5, match is substituted with $(\text{match} + 1)$. Hence, $Q \equiv (\text{match} + 2 == 1)$.
- 3) At assignment node 3, match is substituted with 0. Hence, $Q \equiv (2 == 1) \equiv \text{false}$.

Pattern 1 – Flag-Based Inhibitor Pattern. Let π be a path that passes through a flag-based predicate node f in a CFG. If the predicate of the branch at f in π is evaluated as unsatisfiable by l -constant propagation, then π is an infeasible path matching the **flag-based inhibitor pattern**.

The above-mentioned path $\pi = (\text{entry}, 1, 3, 4, 5, 6, 7, 8, 10, 12, 13, \text{exit})$ in Figure 4-1b traverses flag-based predicate node 12 (node 12 is a flag-based predicate node as it only references to match —a flag). As evaluated earlier using l -constant propagation, the predicate of the branch at this flag-based predicate node in π is $Q \equiv (\text{match} == 1) \equiv \text{false}$. Therefore, π is an infeasible path complying with the flag-based inhibitor pattern.

As a further illustration, in Figure 4-1b, both $(\text{entry}, 1, 2, \text{exit})$ and $(\text{entry}, 1, 3, 4, 6, 8, 10, 11, \text{exit})$ do not match the flag-based inhibitor pattern. The only predicate node that the first path traverses is node 1 that does not refer to any flag. Hence, the path does not traverse any

flag-based predicate node, and therefore, it clearly does not match the flag-based inhibitor pattern. This can be determined solely using syntax analysis. Though the second path traverses flag-based predicate node, *l0*, applying *l*-constant propagation, the predicate of the branch at node *l0* in this path is $Q \equiv (match == 0) \equiv (0 == 0) \equiv true$. Therefore, the second path also does not match the flag-based inhibitor pattern.

Symbolic evaluation is a form of static program analysis in which symbolic expressions are used to denote the values of program variables and computations [107]. In detecting infeasible paths of conflicting branch pattern, a simplified form of symbolic evaluation is applied. In this symbolic evaluation, the joint satisfiability of the predicates of selected branches in a path is evaluated using one of the following two options if suitable:

- 1) **Direct use of SMT Solver:** Use an SMT solver for linear arithmetic constraints to evaluate it directly.
- 2) **Augmented use of SMT Solver through Transformation:** First, include addition predicates to specify the constraints on the return values of mathematical system function-calls involved (if any). Next, transform the predicates into linear constraints by replacing expressions that do not share common variables and do not contain side-effect function-calls with free real variables. Then, evaluate the resulting linear constraints using an SMT solver for linear arithmetic constraints.

If the joint satisfiability cannot be evaluated by these options, then it will be deemed beyond the scope of this symbolic evaluation and it will not be dealt with. We call this simplified form of symbolic evaluation, **limited symbolic evaluation** (*l*-symbolic evaluation). Note that if the outcome from the second option is unsatisfiable, then, clearly, the original predicates can never be satisfied jointly according to the SMT solvers for linear arithmetic constraints. Hence, the soundness of unsatisfiability based on *l*-symbolic evaluation entirely depends on the soundness of the used SMT solver for linear arithmetic constraints.

Let *p* and *q* be two predicate nodes in a CFG. If both *p* and *q* transitively reference a common input variable, then we say that the predicate nodes *p* and *q* are **correlated**. In the CFG shown in Figure 4-2b, node 2 and node 5 are correlated as both of them transitively reference to the input variable *listC* received at the entry node.

Figure 4-2. First conflicting branch pattern

Let π be a path in a CFG that passes through a pair of correlated predicate nodes p and q . Let $\Psi_{p,q} = \{x \mid x \text{ is a predicate node in the sub-path from } p \text{ to } q \text{ in } \pi, x \neq p, x \neq q, \text{ and either } p \text{ and } x \text{ or } q \text{ and } x \text{ both transitively reference to a common input variable}\}$. Any set that contains p and q and a subset of $\Psi_{p,q}$ (including empty subset) is called a **set of weakly correlated predicate nodes** between p and q in π .

Pattern 2 – Conflicting Branch Pattern. Let π be a path that passes through a pair of correlated predicate nodes p and q in a CFG. If l -symbolic evaluation evaluates that the predicates of branches at the predicate nodes in a set of weakly correlated predicate nodes between p and q in π , are unsatisfiable jointly, then π is an infeasible path matching the **conflicting branch pattern**.

In Figure 4-2b, nodes 2 and 5 are correlated predicate nodes as they both transitively reference to the input variable *listC*. Path $\pi = (2, 3, 4, 5, 6)$ passes through these two correlated predicate nodes. $\{2, 3, 5\}$ is a set of weakly correlated predicate nodes between nodes 2 and 5 in π . The predicates of the branches at node 2, 3 and 5 in π are $(listC.size() > 0)$, $(listA.size() > listC.size())$ and $(listA.size() + y < listC.size() + 3)$, respectively. Applying l -symbolic evaluation to evaluate the joint satisfiability of these predicates, we propagate them backward, so as to finally obtain $(listC.size() > 0)$, $(listA.size() > listC.size())$ and $(listA.size() + 3 < listC.size())$. As the expressions *listA.size()* and *listC.size()* in the resulting predicates do not share common variables and also do not have side-effect function calls (from the list of side-effect function-calls maintained manually, we know that *size()* does not have side-effect), therefore, following the second option in l -symbolic evaluation, we replace them with free variables x and z , respectively. After the replacement, the resulting predicates are linear

constraints $(z > 0)$, $(x > z)$ and $(x + 3 < z)$. Clearly, these linear constraints are unsatisfiable jointly. Hence, π is an infeasible path complying with the conflicting branch pattern.

Next, we give an example of an infeasible path complying with conflicting branch pattern in an inter-procedural CFG with a loop. Figure 4-3a shows a customized procedure, *process*, which calls a reusable procedure, *analyseB*, in a library to decode *buffer*. Figure 4-3b shows the inter-procedural CFG of *process*. In this inter-procedural CFG, path $\pi = (a_4, a_5, \text{entry_analyseB}, b_1, b_2, b_3, b_4, b_6, b_7, b_9)$ traverses a pair of correlated predicate nodes a_4 and b_6 . The predicates of the branches at the predicate nodes in the smallest set $\{a_4, b_6\}$ of weakly correlated predicate nodes between a_4 and b_6 in π are $(\text{buffer.length} \geq 0 \ \&\& \ \text{buffer.length} \leq 64)$ and $(\text{length} == 128)$, respectively. Applying *l*-symbolic evaluation to evaluate the joint satisfiability of these predicates, we propagate them backward, and finally obtain $(\text{buffer.length} \geq 0 \ \&\& \ \text{buffer.length} \leq 64)$ and $(\text{buffer.length} == 128)$. As *buffer.length* is the only expression in the resulting predicates and does not have side-effect function-call, therefore, following the second option in *l*-symbolic evaluation, we replace *buffer.length* with free variable x . After the replacement, the resulting linear constraints are $(x \geq 0 \ \&\& \ x \leq 64)$ and $(x == 128)$. Clearly, these linear constraints are unsatisfiable jointly. Hence, π is an infeasible path complying with conflicting branch pattern.

Clearly, any path that contains an infeasible sub-path is also infeasible. Hence, as π has been detected to be infeasible, the infinite family of paths including $(a_2, a_3, a_4, a_5, \text{entry_analyseB}, b_1, b_2, b_3, b_4, b_6, b_7, b_9, \dots, \text{exit_analyseB}, a_5, a_2, a_6, \text{exit_process})$, $(a_2, a_3, a_4, a_5, \text{entry_analyseB}, b_1, b_2, b_3, b_4, b_6, b_7, b_9, \dots, \text{exit_analyseB}, a_5, a_2, a_3, a_4, a_5, \text{entry_analyseB}, b_1, b_2, b_3, b_4, b_6, b_7, b_9, \dots, \text{exit_analyseB}, a_5, a_2, a_6, \text{exit_process})$, and so on, that iterates the loop defined by a_2 with any number of times, are all infeasible. In the proposed approach, once a path contains a sub-path that has been shown as infeasible earlier, the path is concluded as infeasible right away without any further processing.

Figure 4-3. Second conflicting branch pattern

4.2 Detection of Infeasible Paths Using Patterns with Limited Symbolic Evaluation (DIPLE)

In this section, we describe a novel method to infer whether a given path is infeasible through recognizing the two patterns proposed in Section 4.1. The path given can be from intra-procedural or inter-procedural CFG. The underlying assumption, which will be verified by our empirical evaluation in Section 4.3, is that these patterns match a large proportion of infeasible paths in actual systems. Inferring infeasible paths from these two patterns requires only the use of limited symbolic evaluation.

4.2.1 How DIPLE Works

Clearly, if a path does not contain any flag-based predicate node, it will never match with the flag-based inhibitor pattern (Pattern 1). If a path does not contain a pair of correlated predicate nodes, it will never match with the conflicting branch pattern (Pattern 2). Furthermore, flag-based predicate nodes and correlated predicate nodes in a CFG can be identified automatically solely using syntax analysis without any symbolic evaluation. Hence, in the proposed method, before fully checking the compliance of a path with either Pattern 1 or Pattern 2, we first apply syntax analysis to filter away paths that do not contain flag-based predicate nodes and correlated predicate nodes, as these paths will never match with any of the two patterns. By

relying exclusively on syntax analysis, many paths can be processed without resorting to any symbolic evaluation, thus greatly improving the efficiency of the proposed method. This is one key feature of DIPLE.

As the identification of Pattern 1 uses only l -constant propagation, it does not require the use of any constraint solver, thus entailing less processing. Therefore, any path that has not been filtered out by syntax analysis will be first checked against Pattern 1 and then against Pattern 2. For the same reason, when matching against Pattern 2, we always check on the joint satisfaction of fewer branch predicates first. All these heuristics aim at further improving the efficiency of DIPLE. Next, we shall define our terminology before proceeding through the details of DIPLE.

A path in an intra-procedural CFG is **infeasible** if no input data can exercise the path regardless of the result of the associated execution of calling and called procedures. For convenience, in the on-going discussion, intra-procedural CFG and inter-procedural CFG shall be referred to as **intra-CFG** and **inter-CFG**, respectively. Paths in intra-CFG and inter-CFG shall be referred to as **intra-path** and **inter-path**, respectively.

The algorithm *applyDiple* shown in Figure 4-4 shows a framework for applying DIPLE in any context. In this algorithm, given a control flow graph G to be processed, the set $\wp$ of flag-based predicate nodes in G and the set $\Gamma = \{(p, I_p) \mid p \text{ is a predicate node in } G \text{ and } I_p \text{ is the set of input variables that are transitively referenced at } p\}$, are computed and made available to the algorithm *isInfPath* shown in Figure 4-5 that infers whether a given path is infeasible. When a path is generated in *applyDiple*, the algorithm *isInfPath* is invoked to check whether the path is infeasible. A Boolean value is returned by *isInfPath* to indicate whether the path is infeasible. In *isInfPath*, once a path is deemed to match one of the infeasible path patterns and is therefore infeasible, it also identifies a sub-path in the path that also matches the pattern and is therefore also infeasible, and stores it in a set ξ of infeasible sub-paths. All the infeasible paths detected are stored in a set $\mathfrak{I}$. The sets $\wp$, Γ , ξ and $\mathfrak{I}$ are declared as global variables to be used by both algorithms.

Given a path, π , *isInfPath* first checks whether π contains a sub-path in the set ξ (a set of infeasible sub-paths accumulated on the fly). If this is the case, it is deemed infeasible right away without any further processing and included in $\mathfrak{I}$; otherwise, the infeasibility of π is determined as follows:

- 1) First, the set $\wp_\pi$ of flag-based predicate nodes in π is extracted from the set $\wp$ of all flag-based predicate nodes in G , as previously constructed in *applyDiple*. Next, if $\wp_\pi \neq \emptyset$, π is checked against Pattern 1 by invoking the function *lconstPropagPattern1*(π , f)

to perform the following actions, using each time a flag-based predicate node f in $\wp_\pi$ until π has been inferred as infeasible or all the nodes in $\wp_\pi$ have been used:

- Given with two parameters π and f , l -constant propagation is applied to the predicate of the branch at f in π . If l -constant propagation evaluates that the predicate can never be satisfied, then π is deemed matching Pattern 1 and infeasible. If π is infeasible, then clearly the sub-path in π that contains all the nodes used for checking compliance with Pattern 1 also matches the pattern and is identified as an infeasible sub-path in π . The result is returned in two elements, *isInfeasible* and *infeasibleSubPath*, the former being a Boolean value indicating whether π is infeasible and the latter being the identified infeasible sub-path in π .

Finally, if π is inferred as infeasible, then π is included in $\mathcal{T}$ and the returned infeasible sub-path in π is included in ξ .

- 2) If π has not been inferred as infeasible, the set $\mathcal{R}$ of pairs of correlated predicate nodes in π is identified by checking the intersection between the sets of input variables transitively referenced for each pair of predicate nodes in π . The latter sets are obtained from the set Γ , as previously constructed in *applyDiple*. Next, if $\mathcal{R} \neq \emptyset$, π is checked against Pattern 2 by invoking the function *lSymEvalPattern2* to perform the following actions, using each time a pair (p, q) of correlated predicate nodes in $\mathcal{R}$ and a weakly set W of correlated predicate nodes between them in π , from the smallest set to the largest set, until π has been inferred as infeasible or all the nodes in $\mathcal{R}$ and all their former sets have been used:

- Given the two parameters π and W , l -symbolic evaluation is applied to evaluate the joint satisfiability of the predicates of branches at the nodes in W . If they can never be satisfied jointly, then π is concluded to be an infeasible path matching Pattern 2. If π is infeasible, then clearly the sub-path in π that contains all the nodes used for checking compliance with Pattern 2 also matches the pattern and is identified as an infeasible sub-path in π . The result is returned in two elements, *isInfeasible* and *infeasibleSubPath*, the former being Boolean value indicating whether π is infeasible and the latter being the identified infeasible sub-path in π .

Finally, if π is inferred as infeasible, then π is included in $\mathcal{T}$ and the returned infeasible sub-path in π is included in ξ .

Finally, *isInfPath* returns a Boolean value to indicate whether π is infeasible.

When applying DIPLE, except for standard program analysis, no additional, specific processing is required to deal with the difference between intra-procedural and inter-procedural CFG.

```

Input:          G: the CFG of a procedure;
Output:         -----
Global Variables:  $\wp$ : the set of flag-based predicate nodes in G;
                    $\Gamma$ : the set  $\{(p, I_p) \mid p \text{ is a predicate node in } G \text{ and } I_p \text{ is the set of}$ 
input variables that are transitively referenced at  $p\}$ ;
                    $\xi$ : a set of infeasible sub-paths accumulated;
                    $\mathfrak{F}$ : a set of infeasible paths in G
Algorithm applyDiple (G;  $\Omega$ )
begin
1.   $\Gamma = \emptyset, \wp = \emptyset, \xi = \emptyset, \mathfrak{F} = \emptyset$ ;
    // compute  $\wp$  and  $\Gamma$ ;
2.  for each predicate node  $p$  in  $G$  do
3.    if  $p$  is a flag-based predicate node then
4.      include  $p$  in  $\wp$ ;
    endif
5.     $I_p = \{v \mid p \text{ transitively references to the input variable } v\}$ ;
6.    include  $(p, I_p)$  in  $\Gamma$ ;
    endDo

/*implement the application main function in which once a path is generated, it is
submitted to algorithm shown in Figure 4-5 to infer its infeasibility */
    While (function uncompleted) do
        -----
        -----
n.    generate a path  $\pi$ ,

    // call the algorithm shown in Figure 4-5 to infer the infeasibility of  $\pi$ 
n+1. isInfPath(G;  $\pi$ );
        -----
        -----
    endWhile
end

```

Figure 4-4. A framework algorithm to apply DIPLE

Input: G : the CFG of a procedure;
 π : a path;
Output: $isInfeasible$: a boolean value to tell whether the π is infeasible
Algorithm ***isInfPath***(G ; π)

```

begin
1.    $isInfeasible = \text{false}$ ;
// inferring infeasibility from its containing of infeasible sub-path
2.   if  $\pi$  contains a sub-path in  $\xi$  then
3.      $isInfeasible = \text{true}$ ;
4.     include  $\pi$  in  $\mathfrak{F}$ ;
    endIf

5.   if not  $isInfeasible$  then
// inferring infeasibility from Pattern 1
6.      $\wp_\pi = \{p \mid p \text{ is in } \pi, p \in \wp\}$ ;
7.     while (not  $isInfeasible$  and  $\wp_\pi \neq \emptyset$ ) do
8.        $f = \text{an element in } \wp_\pi$ ;
9.        $\wp_\pi = \wp_\pi - \{f\}$ ;
10.       $(isInfeasible, infeasibleSubPath) = lconstPropagPattern1(\pi, f)$ ;
11.      if  $isInfeasible$  true then
12.        include  $\pi$  in  $\mathfrak{F}$ ;
13.         $\xi = \xi \cup \{infeasibleSubPath\}$ ;
      endIf
    endWhile
  endIf

14.  if not  $isInfeasible$  then
// inferring infeasibility from Pattern 2
15.     $\mathcal{R} = \{(p, q) \mid p \text{ and } q \text{ are in } \pi \text{ in the order shown, } (p, I_p) \in \Gamma, (q, I_q) \in \Gamma, I_p \cap I_q \neq \emptyset\}$ ;
16.    while (not  $isInfeasible$  and  $\mathcal{R} \neq \emptyset$ ) do
17.       $(p, q) = \text{an element in } \mathcal{R}$ ;
18.       $(isInfeasible, infeasibleSubPath) = lSymEvalPattern2(\pi, \{p, q\})$ ;
19.       $\mathcal{R} = \mathcal{R} - \{(p, q)\}$ ;
20.      if not  $isInfeasible$  then
21.         $\Psi_{p,q} = \{x \mid x \text{ is a predicate node in the sub-path from } p \text{ to } q \text{ in } \pi, x \neq p, x \neq q, \text{ and either } p \text{ and } x \text{ or } q \text{ and } x \text{ both transitively reference to a common input variable}\}$ ;
22.         $W = \{p, q\}$ ;
      endIf
23.      while (not  $isInfeasible$  and  $\Psi_{p,q} \neq \emptyset$ ) do
24.         $x = \text{a predicate node in } \Psi_{p,q}$ ;
25.         $\Psi_{p,q} = \Psi_{p,q} - \{x\}$ ;
26.         $W = W \cup \{x\}$ ;
27.         $(isInfeasible, infeasibleSubPath) = lSymEvalPattern2(\pi, W)$ ;
      endWhile
28.      if  $isInfeasible$  then
29.        include  $\pi$  in  $\mathfrak{F}$ 
30.         $\xi = \xi \cup \{infeasibleSubPath\}$ ;
      endIf
    endWhile
  endIf
31.  return  $isInfeasible$ 
end

```

Figure 4-5. An algorithm to infer the infeasibility of a path

4.3 Empirical Evaluation

We conducted a large-scale study to evaluate DIPLE empirically based on Java code. Java code was chosen as a focus due to its widespread usage. As DIPLE detects infeasible paths based on our proposed infeasible path patterns (Section 4.2), this evaluation also indirectly evaluates how widespread such patterns are in practice.

4.3.1 Objectives and Dependent Variables

The objectives are to evaluate the *correctness*, *completeness*, and *efficiency* of DIPLE. We evaluate these by comparing DIPLE with the results obtained from evaluating path infeasibility exhaustively using symbolic evaluation. In the latter evaluation, we use a state-of-the-art constraint solver Z3 as an SMT solver. For easier reference, in the following discussion, the latter evaluation shall be referred as **exhSymbEvaPhInf**.

In this evaluation, we do not consider the inaccuracy caused by floating-point rounding in existing SMT solvers though this will be separately addressed in Section 4.5.

Correctness is evaluated by comparing DIPLE with exhSymbEvaPhInf on the percentages of paths that are wrongly deemed to be infeasible. Completeness is evaluated by comparing the two methods on the percentages of infeasible paths not detected.

State-of-the-art, precise methods for detecting infeasible paths need to apply symbolic evaluation. Symbolic evaluation is much more expensive than syntax analysis and if DIPLE can use syntax analysis to process a sizable proportion of the paths without any symbolic evaluation, then its efficiency will be greatly improved. Hence, for the evaluation of efficiency, we examine the percentage of paths that are filtered by DIPLE using only syntax analysis. In addition, we also compare the processing time per path by the two methods.

4.3.2 Experiment Design and Analysis

We designed the following two-stage experiment to compare DIPLE and exhSymbEvaPhInf:

Stage One — Reconciliation of Infeasible Paths Detected between DIPLE and exhSymbEvaPhInf: For evaluating the correctness and completeness, in the first stage, we compared the infeasible paths detected between DIPLE and exhSymbEvaPhInf. In stage one, in order to evaluate the *efficiency* of DIPLE; we also compared average processing time per path taken by the two methods

Stage Two — Investigating the Differences found in Stage One: In the second stage, we studied and analyzed the additional infeasible paths detected by any of the methods.

For the generality of our evaluation results, we have to address three aspects: path structure, application domain, and level of system maturity. Regarding path structure, we organize our evaluation into three experiments accounting for, sets of paths extracted according to two commonly used path coverage criteria and also sets of paths extracted randomly as follows:

- 1) **BP0 Experiment (Basis Path Coverage with 0 Loop Iteration Experiment):** This evaluates on basis sets of linearly independent paths through the CFGs of sampled code with no loop iteration [145, 168].
- 2) **CtP2 Experiment (Ct Path Coverage with 2 Loop Iteration Experiment):** This evaluates on sets of paths that provide Ct coverage with $k = 2$ [32] — each of these sets contains all the paths through the CFGs of sampled code with *at most* two loop iterations.
- 3) **Random Experiment:** This is to evaluate on a set of 10,001 paths through the CFGs of sampled code extracted randomly but according to proportions driven by the cyclomatic complexities [168] of CFGs. In the extraction, a random number is generated to decide on the branch and number of loop iterations to follow upon encountering each selection and iteration construct respectively.

We conducted the three experiments on intra-CFG and inter-CFG separately. For the former, we extracted the paths through the intra-CFGs of all the procedures in the sampled code. For the latter, we extracted the paths through the inter-CFGs of all the root procedures in the sampled code.

To address the generality of our evaluation results across application domains, we selected a sizable number of open-source systems from a variety of application domains [5, 8, 11, 18]. Regarding level of system maturity, in addition to open-source systems, we also selected industrial systems and systems with seeded faults from the infrastructure for software testing research [61].

4.3.3 Automation

We have developed a prototype system to implement DIPLE, exhSymbEvaPhInf and the extraction of paths for the three experiments. Our prototype system is running on NEC Powermate, with Intel Duo E6750 2-core processor, 2.66 GHz and 4 GB memory. It is developed under Windows XP using Oracle JDK 1.6.25 and Soot 2.5.

The overall structure of the prototype system is shown in Figure 6. The system is built using Soot [7] and Z3 [17].

Soot is a static analysis framework with many build-in tools for Java programs [111]. Its core is a Java compiler. It accepts Java Virtual Machine bytecode or Java source code. Soot's latest version supports analysis over Java programs with syntax and features of Oracle JDK 8.

The prototype system implements DIPLE according to the algorithm shown in Figures 4 and 5. In the implementation of our *l*-symbolic evaluation in DIPLE, we use Z3 as a constraint solver. Z3 is a theorem prover [15] with high performance. It supports solving constraints over large types of operations, including linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, un-interpreted functions, quantifiers, etc. [17].

The system implements `exhSymbEvaPhInf` that follows the way of a state-of-the-art symbolic execution tools [162]: for each traversed path, it uses exhaustive symbolic evaluation to generate a path constraint. The path constraint is sent to equipped constraint solver to evaluate path infeasibility. Here, we also use Z3 as such constraint solver.

For each selected testing program, the prototype system prepares its source code and required libraries in advance. Soot is used to construct intra-CFGs [68] and inter-CFGs [111]. The prototype system extracts paths for BP0 [145], CtP2 [32] and Random [168] experiments separately, according to request.

The system first generates a set of paths and runs DIPLE and `exhSymbEvaPhInf` over this path set independently. The infeasible paths identified by the two methods are compared and analyzed to generate a final report.

Figure 4-6. The structure of our prototype system

4.3.4 Experiment Results

The experiment was carried on a large set of systems with wide variation in path size. For the experiments on intra-paths, the average numbers of nodes per path are 22.74, 36.64 and 136.88 for BP0, CtP2 and Random experiments, respectively. For the experiments on inter-paths, the average numbers of nodes per path are 26.41, 63.81 and 218.72 for BP0, CtP2 and Random experiments, respectively. Table 4-1 summarizes the statistics on the size of the experimented systems. The detailed results can be obtained from our website [60].

Table 4-1. Systems for experiment

Experiment	Number of Systems	Total Paths	#Node/path	
			Mean	Stdev
BP0 experiment on intra-paths	16	55,383	22.94	43.54
BP0 experiment on inter-paths	16	44,579	26.41	53.29
CtP2 experiment on intra-paths	8	328,575	36.64	52.47
CtP2 experiment on inter-paths	8	693,885	63.81	62.74
Random experiment on intra-paths	8	10,001	136.88	333.27
Random experiment on inter-paths	8	10,001	218.72	745.91

Stage One — Reconciliation of Infeasible Paths Detected between DIPLÉ and exhSymbEvaPhInf

To evaluate the *correctness* and *completeness* of DIPLÉ, we compared the infeasible paths detected between DIPLÉ with exhSymbEvaPhInf in the first stage of our experiment. The following summarizes the key results:

- All the infeasible paths detected by exhSymbEvaPhInf are also detected by DIPLÉ.
- In addition, DIPLÉ detects 23.82% and 21.94% more infeasible paths on intra-paths and inter-paths, respectively in the BP0 experiment; 26.86% and 27.04% more infeasible paths on intra-paths and inter-paths, respectively in the CtP2 experiment; 29.50% and 22.50% more infeasible paths on intra-paths and inter-paths, respectively in the Random experiment. Overall, DIPLÉ detects 26.69% more infeasible paths. The correctness of on these additional infeasible paths detected by DIPLÉ will be investigated later in the stage two.

Figure 4-7 to Figure 4-12 compare the infeasible paths detected between DIPLÉ and exhSymbEvaPhInf in the various systems. The comparison is shown in the form of histograms. In these histograms, the grey areas show the infeasible paths detected by both DIPLÉ and exhSymbEvaPhInf. The white areas show the additional infeasible paths detected by DIPLÉ alone.

To evaluate the efficiency of DIPLE, we compared average processing time per path taken by the two methods. DIPLE is always faster than exhSymbEvaPhInf in all the experiments. Figure 4-13 to Figure 4-18 compare the processing times per path of DIPLE with exhSymbEvaPhInf, side-by-side for each system in the experiments. The light grey areas stand for the processing time per path taken by DIPLE. The areas filled with oblique lines stand for the processing time per path taken by exhSymbEvaPhInf. On average, DIPLE is faster by 54.65% in the BP0 experiment for intra-paths and is faster by 78.32% in the BP0 experiment for inter-paths. DIPLE is faster by 58.52% in the CtP2 experiment for intra-paths and is faster by 59.62% in the CtP2 experiment for inter-paths. DIPLE faster by 44.73% in the Random experiment for intra-paths and is faster by 72.01% in the Random experiment for inter-paths.

Figure 4-7. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in BP0 experiment on intra-paths

Figure 4-8. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in BP0 experiment on inter-paths

Figure 4-9. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in CtP2 experiment on intra-paths

Figure 4-10. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in CtP2 experiment on inter-paths

Figure 4-11. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in Random experiment on intra-paths

Figure 4-12. Comparison of infeasible paths detected between DIPLE and exhSymbEvaPhInf in Random experiment on inter-paths

Figure 4-13. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in BP0 experiment on intra-paths

Figure 4-14. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in BP0 experiment on inter-paths

Figure 4-15. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in CtP2 experiment on intra-paths

Figure 4-16. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in CtP2 experiment on inter-paths

Figure 4-17. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in Random experiment on intra-paths

Figure 4-18. Comparison of average processing time per path between DIPLE and exhSymbEvaPhInf in Random experiment on inter-paths

Stage Two — Investigating the Differences found in Stage One

As all the infeasible paths detected by exhSymbEvaPhInf are also detected by DIPLE, in stage two, we investigated the additional infeasible paths detected by DIPLE. Through much extensive effort with manual checking, we managed to classify the reasons that why DIPLE managed to detect these paths into two:

- 1) Due to the substitution of expressions of function calls and non-linear operations by 1-symbolic in DIPLE (called **replacement**): Infeasible paths detected through such substitution cannot be detected by exhSymbEvaPhInf due to the predicates of branches of these paths involve non-linear operations or calling of functions that do not Java source and JIMPLE code. In BP0 experiment, with such replacement, DIPLE detected 7.90% additional infeasible intra-paths (this is the percentage of infeasible paths detected by DIPLE in the total number of infeasible paths detected) and 14.91% additional infeasible inter-paths. In CtP2 experiment, DIPLE detected 26.11% additional infeasible intra-paths and 26.70% additional infeasible inter-paths. In Random experiment, DIPLE detected 29.17% additional infeasible intra-paths and 21.94% additional infeasible inter-paths.
- 2) Due to the abstraction by the two proposed infeasible path patterns (called **abstraction**): As exhSymbEvaPhInf exhaustively evaluates the conditions of all branches, it encounters runtime exceptions or timeout for some paths that have many branches, and consequently the path infeasibility cannot be evaluated. The two proposed infeasible paths patterns allow DIPLE to evaluate only conditions of limited branches (usually one or two) for such paths. This enables DIPLE to evaluate the infeasibility of such paths. In BP0 experiment,

DIPLE does not detect any additional infeasible intra-paths due to this abstraction, but this abstraction helps to detect 0.25% infeasible inter-paths. In CtP2 experiment, this abstraction helped to detect 0.75% additional infeasible intra-paths and 0.34% infeasible inter-paths. In Random experiment, this abstraction helped to detect 0.33% additional infeasible intra-paths and 0.56% infeasible inter-paths.

Figure 19 to Figure 24 show the classification of additional infeasible paths detected by DIPLE using histograms for all the sample systems in the experiments.

Figure 4-19. Reconciliation of infeasible paths detected in DIPLE and exhSymbEvaPhInf in BP0 experiment on intra-paths

Figure 4-20. Reconciliation of Infeasible Paths Detected in DIPLE and exhSymbEvaPhInf in BP0 experiment on inter-paths

Figure 4-21. Reconciliation of Infeasible Paths detected in DIPLE and exhSymbEvaPhInf in CtP2 experiment on intra-paths

Figure 4-22. Reconciliation of infeasible paths detected in DIPLE and exhSymbEvaPhInf in CtP2 experiment on inter-paths

Figure 4-23. Reconciliation of infeasible paths detected in DIPLE and exhSymbEvaPhInf in Random experiment on intra-paths

Figure 4-24. Reconciliation of infeasible paths detected in DIPLE and exhSymbEvaPhInf in Random experiment on inter-paths

4.3.5 Threats to Validity

Though we have compared DIPLE with exhSymbEvaPhInf and also carefully analyzed the comparison results, the following issues may still pose threats to validity:

- 1) Though Z3 is a state of the art constraint solver, it still cannot deal with floating point arithmetic accurately. Hence, the fact that DIPLE detected not only all the infeasible paths detected by exhaustive symbolic evaluation using Z3, but also more infeasible paths that were also semi-automatically checked using Z3 and found to be correct, does not mean that all these infeasible paths are 100% truly infeasible paths. This is because of the floating point error in Z3.
- 2) Though the scale and scope of our study are quite extensive, there is still room for further improvements regarding generality, including more randomly selected paths and industry systems, and covering more levels of system maturity (pre-unit-testing stage, operational stage, etc.).
- 3) In order to perform an evaluation of DIPLE on systems written in a different programming language, we need to develop a different prototype analysis tool. Due to the substantial effort involved in such development, all our experiments were performed on Java code. Though there is no reason to believe that the proposed infeasible path patterns and the principles on which DIPLE relies depend on any specific programming language, replications of our studies with different programming languages are still required as, for example, the usage of floating-point computation might be different among languages.

4.4 Applications

Many software engineering techniques including structural testing, code optimization, etc., have to deal with program paths. In these techniques, the ability to detect infeasible paths is useful for improving either their precision or effectiveness. Based on the algorithms shown in Figure 4-4 and Figure 4-5 for general use structure, DIPLE can be instrumented in any of these techniques to detect infeasible paths. In the remaining of this section, we shall give some details on a couple of most direct application of DIPLE to serve as examples.

Structural test coverage for a test suite, given a specified coverage criterion, is computed as the percentage of program elements exercised given all program elements selected according to the coverage criterion. It is a fundamental and important principle for test adequacy and therefore to assess the quality of a test suite. One long-standing practical issue is that the existence of infeasible program elements seriously reduces the precision of structural test coverage [29], to the extent that can prevent its practical usage.

The most direct application of DIPLE is in the computation of structural test coverage for path-based coverage criteria. It helps improve its precision by excluding most of the infeasible paths in the computation. As the inference of most infeasible elements in a program can be deduced from the inference of path infeasibility, DIPLE can also be used to improve the precision of most other structural test coverage computations to the same extent.

Another application is to instrument DIPLE in the newer generation of test case generators, such as CUTE, PEX and PathCrawler [151, 162, 173], for precisely computing the coverage achieved against the coverage criteria for a test suite generated. Also, in such instrumentation, the time of symbolic evaluation upon encountering an infeasible path can be reduced as DIPLE can detect it more quickly.

4.5 Conclusion

Our goal was to improve the current state of the art by making the detection of infeasible paths in real, large-scale systems more accurate and efficient. The main contributions of this chapter include:

- 1) We have identified and defined two patterns based on CFGs of real system code that lead to infeasible paths: the flag-based inhibitor pattern and the conflicting branch pattern. We proposed the DIPLE method that relies on detecting these patterns in an efficient way to identify infeasible paths.
- 2) In our extensive experiments on both intra and inter-CFGs of the Java code of many diverse systems, when excluding floating-point rounding problem in Z3, all the infeasible paths detected by DIPLE were truly infeasible. Furthermore, DIPLE detected more

infeasible paths than exhaustively using symbolic evaluation. The experiments also demonstrated that DIPLE was more efficient and could therefore scale to large systems. The experiment results suggest that DIPLE can be applied in practice in the context of industrial systems and for various applications, including coverage measurement.

We believe that our way of addressing the infeasible path problem—characterizing the target objects (infeasible paths in our case) in real systems to overcome theoretical limits (constraints are in general intractable) such that a practical solution can be found—could be an important direction to address other issues such as security vulnerabilities and fault detection, in software analysis and testing.

Chapter 5

DETECTION OF BUFFER OVERFLOW VULNERABILITIES IN C/C++ WITH PATTERN BASED RESTRICTED SYMBOLIC EVALUATION

Buffer overflow vulnerability is one of the serious security threats for applications written in C/C++. Among existing buffer overflow detection methods, path sensitive approach has higher accuracy. Along a single path, it performs symbolic evaluation to capture the impact of variables that interact with buffers and find out whether such impact could violate existing buffer boundaries [113, 115]. However, existing path sensitive approaches include the following limitations:

- 1) Existing methods traverse each path to generate constraints. If the path is long, there is a higher risk to meet unsolvable constraints. For example a recent method, Marple [113] [114], would raise a constraint and start to do symbolic evaluation along the path when it encounters suspicious operations. When encountering a node that defines a variable in the constraint, their method would evaluate the constraint immediately. Therefore during the propagation, once there is a node that causes the constraint unsolvable, the evaluation has to terminate. However, there is a possibility that the partial constraint from the rest nodes is enough to determine the path is vulnerable.
- 2) Certain buffer overflow vulnerabilities could be identified with only checking the syntax rather than using symbolic evaluation. For example, in Figure 5-1, the function *customizedPrint()* contains a vulnerability at node4. It is not necessary to capture such case by using heavy symbolic evaluation.
- 3) Existing path sensitive methods might cause considerable overhead. If the path is long, there might be a large number of symbolic representations during the path propagation, or if the control flow graph has complex control structure like nested loops, there might be a large number of paths to process. For example, in Figure 5-3, there would not be an effective way to identify the vulnerability at node7 unless the value range of variable *i* could be captured properly. Though Saxena [150] proposes a method to handle loop, it only considers variables that are updated linearly within each iteration.

Therefore, we propose a method aiming to solve the above limitations. The method is to identify paths with buffer overflow vulnerability by pattern recognition. We present three code patterns in Section 5.1.1. Pattern recognition consists of syntax analysis and a restricted form of symbolic evaluation, which is in Section 5.1.2. The main contributions in this chapter include:

- 1) We improve the scalability of buffer overflow detection for a given path through a method of pattern recognition. Pattern recognition is scalable because it relies on a light weighted analysis that consists of syntax analysis plus a restricted form of symbolic evaluation.
- 2) We improve the scalability of buffer overflow detection for a given program because we use the relationships between detected vulnerable paths and the others waiting for detection.
- 3) We do not scarify detection accuracy because the proposed method is path sensitive, loops and aliased structures are also within the consideration.

5.1 Buffer Overflow Patterns with Symbolic Evaluation

5.1.1 Code patterns

We propose three empirical patterns to cover the three possibilities causing programs vulnerable against buffer overflow attacks in real codes: pattern of vulnerable syntax usage, pattern of vulnerable element accessing and pattern of vulnerable bulk moving.

We have adopted the formalism of control flow graph (CFG) from Sinha et al [155]. Furthermore we define that, in a CFG, a buffer overflow path sensitive sink is a statement in a program such that the execution of it may lead to buffer overflow if there is no validation to restrict the values of variables referenced at this statement properly. Correspondingly, the node that represents a buffer overflow path sensitive sink is called a buffer overflow path sensitive sink node. For convenience, we abbreviate it as **sink node** in the following part. We consider variables from function parameters, database, files or command-line all as input variables.

C/C++ is a language with flexible syntax that assigns users high convenience to develop programs with high performance. However on the negative side, the flexibility results in many codes that are not carefully written and exploitable for attacks. Heuristically, we only consider the situations with string format functions since they lead to the majority of the buffer overflows [9]. String format function in C is such a case. They appear as symbols (e.g.: `%s` or `%d`) and function as one parameter. They control the processing and displaying of other string functions. The secure way of using such string format functions is to put the string

content after the string format symbol, such as `printf ("%s", input_str)`, where `input_str` is a user input. However, C/C++ allows code written like `printf(input_str)`. Hackers could launch an attack by purposely inserting the string format symbols like `%s` inside of `input_str` to access memory illegally [116].

Though the attacks may consist of complex techniques, we found that quite a number of the buffer overflow vulnerabilities are caused due to using C/C++ syntax in a vulnerable way. These cases could be detected by pattern1.

Pattern1 – Pattern of Vulnerable Syntax Usage. A sink node k is threatened by buffer overflow attack due to improper syntax usage, when either of the following is detected along a path:

- 1) k references one C string format function sf , and before k there is an assignment node assigning an input variable to one of the sf 's parameter, where compiler would misunderstand the input variable as a string format symbol [116].
- 2) k references a buffer as one parameter of those C functions which are relying on end characters. And before k there is assignment node assigning an input variable to the last buffer element.

The assignment node in either of the two situations above is called the vulnerable root node (denoted as vrn) of pattern1.

```
void customized_print( ) {
/*1*/  char tmpbuf[512];
/*2*/  char format[30];
.....
/*3*/  scanf( "%s", format);
/*4*/  char * info = read_user (tmpbuf, 512);
.....
/*5*/  printf (format, info);
}
```

Figure 5-1. Pattern of vulnerable syntax usage

This pattern can be automatically identified easily with syntax analysis: once a sink node with vulnerable syntax usage is found on a path. The pattern recognition procedure first propagates backward along the path to confirm whether any assignment nodes could be the corresponding vrn . The procedure will terminate when it did not find any vrn . Otherwise, the procedure considers that it identifies an instance of pattern1. In Figure5-1, the function `customized_print( )` is for users to customize the output format of their information. The sink node locates at node5. The backward propagation first locates the assignment at node3. Syntax analysis confirms node3 is the corresponding vrn . Thus, an instance of pattern1, which consists of node3 and node5, is identified along the path (node1, node2, node3, node4, node5). Therefore, this path is identified as vulnerable against buffer overflow attacks.

Besides those syntax errors expressed by pattern1, there are other violations also could cause buffer overflow vulnerability. We model such violations using another two patterns.

Pattern2 – Pattern of Vulnerable Element Accessing. A sink node k is threatened by buffer overflow attack if k assigns an input variable to a buffer element and none of the following occurred:

- 1) Before k , there is an assignment node defining the element's index, through symbolic evaluation, it is proved the index's current definition is within the buffer's boundary.
- 2) Before k , there is (are) predicate node(s) checking the element's index. Through symbolic evaluation, it is proved the predicate condition restricts the index within the buffer's boundary.

The assignment node and predicate node mentioned in the above two cases are called the vulnerable root node (denoted as vrn) of pattern2.

```
void static char * encode (char * buf, int buf_size, char * p){
    /*1*/ for( int i=0; i< buf_size && i>2; i++) {
        .....
        /*2*/     if ( *p=='n')
        /*3*/         buf[i] = *p;
        /*4*/     if( *p=='x')
        /*5*/         buf [ i + 1] = 'o';
        } //end for
        .....
        /*6*/ char * info = buf;
        .....
        /*7*/ return info;
    }
```

Figure 5-2. Pattern of Vulnerable Element Accessing

This pattern can be automatically identified with syntax analysis and symbolic evaluation: once the sink node of pattern2 is found, the pattern recognition procedure first tries to locate the corresponding vrn backward along the path. This procedure terminates if it does not find any vrn . Otherwise, symbolic evaluation would be involved to verify whether the vrn could allow the element's current index within the buffer's boundary. If it found the index is out of the boundary, an instance of pattern2 is identified, and this path is evaluated as vulnerable. We demonstrate patten2 with an example followed a similar case in CVE-2006-6332 for system MadWifi-0.9. In Figure 5-2 node1 is the predicate node performing checking the value of variable i and it ensures the element accessing at node3 is safe. However through the identification of patter2, we will see, the node5 is vulnerable because the index $i+1$ will exceed the buf 's boundary when $i = buf_size$.

Pattern3 – Pattern of Vulnerable Bulk Moving. A sink node k is threatened by buffer overflow attack if k uses C string functions to move a bulk of buffer elements from a source

buffer to a destination buffer, and the source buffer is data dependent on input variable. Meanwhile, none of the following could be found:

- 1) Before k , there is an assignment node defining the length of the copied content, through symbolic evaluation, the length is proved to be smaller than the sizes of both source buffer and destination buffer.
- 2) Before k , there are predicate nodes referencing the length of the copied content. Through symbolic evaluation, it is proved the constraint generated by joining these predicates' conditions restricts the length smaller than the sizes of both source buffer and destination buffer.

Similarly, the assignment node or predicate nodes mentioned in the above two cases are called the vulnerable root nodes (denoted as *vrn*) of pattern3.

This pattern can be automatically identified with syntax analysis and symbolic evaluation: once the sink node of pattern3 is found, the pattern recognition procedure tries to locate the *vrn* backward along the path. This procedure terminates if it does not find any *vrn*. Otherwise, the procedure will generate constraint based on the assignment node or predicate nodes. The constraint is then checked by symbolic evaluation and constraint solver. If the constraint is found unsatisfiable, an instance of pattern3 is identified, and this path is evaluated as vulnerable.

We illustrate this process through an example shown in Figure 5-3, from CVE-2006-6909 for the system of Edbrowse-3.1.3. In this example, node7 is a sink node in the path (node1, node2, node3, node4, node5, node6, node7); because it used the vulnerable C string function *strcpy* which has no build-in protection to prevent moving a chop of memory to a smaller destination. The abstract syntax tree of node7 consists of two nodes, the first node defining a temporary variable $x = line + j$; and the second node is *strcpy(user, x)*. Here for convenience, we just consider node7 as the sink node and the *vrn* at the same time. Through proper symbolic evaluation, it is found the constraint: $user.length \geq (line.length - j)$ cannot be satisfiable, which means the copied content cannot be guaranteed smaller than the source buffer. Thus, an instance of pattern3 is found on the path (node1, node2, node3, node4, node5, node6, node7), and this path is identified as vulnerable.

```

void ftpls (char *line, int USERSZ)
{
/*1*/  char user[USERSZ];
/*2*/  int j;
/*3*/  for(j = 0; line[j] != EOS; ++j)
/*4*/    if (line[j] == "-")
/*5*/      break;

/*6*/  if(line[j] == ' ')
/*7*/    strcpy (user, line + j);
}

```

Figure 5-3. Pattern of vulnerable input validation

5.1.2 Restricted Symbolic Evaluation

During the recognition of pattern2 and pattern3, the symbolic evaluation we applied is a restricted form which only evaluates selected nodes. The term “restricted form” means that only typical kinds of paths, variables and operations that satisfy the syntax definitions in pattern2 and pattern3 will be further processed with symbolic execution and constraint solving. The evaluation is to propagate backwards along the path, substituting the definition of variables, into the branch conditions, from selected predicate nodes. The value of such evaluation is to verify the unsatisfiability of constraint generated by pattern2 and pattern3. The evaluation consists of a limited number of operations. It starts when encountering a sink node k . It ends when the joint unsatisfiability is found, or the entry node has been reached. The detailed steps of such evaluation include:

- 1) The evaluation formalizes a constraint according to pattern2 or pattern3 when it encounters a sink node while traversing the path.
- 2) It only keeps track of definitions for integer variables, pointers, arrays transitively referenced by sink node. It updates the existing constraint with these variables’ latest definition.
- 3) It joins existing constraint with encountered predicate conditions by using the logic operator “AND”.
- 4) It does constant propagation. It executes functions listed in `<math.h>` and `<string.h>` when all of the parameters are constants and it replaces the functions’ symbolical representation with the concrete value.

- 5) If the existing constraint contains a function f , the evaluation searches for a mock object that simulates f . The symbolical representation of f will be replaced by the return value of such mock object. Mock object is simulated objects that mimic the behaviour of real program objects. There are a number of mock object frameworks (e.g.: Google C++ Mocking Framework, CUnitWin32) released for industrial applications. However, if no existing mock object available, users will have 2 options:
 - a) The evaluation performs an inter-procedure analysis to obtain the symbolic return value from f with the current parameters. This option has a high accuracy but may produce considerable overhead.
 - b) The evaluation arbitrarily treats f with its actual parameters as an independent variable and replaces it by a free variable. This option has no overhead but may bring in faults.
- 6) If the existing constraint contains other non-linear items like multiplications or divisions, it replaces items that do not mutually share common variables with free variables.
- 7) If it finds that the existing constraint contains only linear and constant items, then submit this constraint to Z3, a state-of-the-art theorem prover, to evaluate the joint unsatisfiability.

The steps (1) to (7) are sufficient for paths that do not pass through any loops and does not contain any aliased pointers or references. However, loops and aliased structures are common in C/C++. Therefore, we propose additional steps to provide support.

For loop handling, existing static analysis methods can be categorized into two. The first method is straightforward as it uses an approximated set of paths to represent loops. Later symbolic evaluation only processes this set of paths. Typical cases can be found in [89, 113, 167]. Such design makes implementation straightforward, but it will cause precision loss. The second approach is to generate abstracted model over loops. Saxena et al [150] proposed a framework to use loop count to represent variables modified within a loop. However it only supports variables that are modified linearly within each iteration. We extend this framework to find out the value range for variables that are non- updated linearly by a loop.

For alias analysis, since variables with different lexical names may represent the same memory areas, a sound method should be able to capture the relationships between these variables precisely. However a sound alias analysis is often very expensive due to complex data structure, such as a function pointer and a doubly indirect pointer. In order to manage the balance between scalability and soundness, we create an additional component, *AliasAnalysis*, to perform path sensitive alias analysis before doing the restricted symbolic evaluation for each

given path in advance. *AliasAnalysis* does not target an entire program. Instead, it tries to get the aliased structures on a path. *AliasAnalysis* is extended from CodeSurfer [9], a state-of-the-art static analysis tool for C/C++. *AliasAnalysis* possesses two parameters: (i) π – the target path in the control flow graph G , (ii) S – the sink nodes on π . *AliasAnalysis* generates ψ , a set of equivalent classes by putting aliased variables along with their referenced locations into the same class. *AliasAnalysis* only considers nodes that are transitively referenced by the sink nodes in S . After this, the precision of the analysis is controlled by CodeSurfer’s build-in precision policy of alias analysis. Therefore, we propose the following steps as a supplement for steps (1) to (7):

- 8) For a given procedure G , the evaluation first finds out all the loop structures and makes each loop structure as a basic block.
- 9) For a given path, before performing steps (1) to (7), the evaluation invokes *AliasAnalysis* to calculate ψ , which is a set of classes, each of which is called an alias class and contains the aliased pointers or references.
- 10) After step (9), the evaluation traverses the path and performs steps (1) to (7) until encountering a node:
 - a) This node is within a loop and contains variables that are transitively referenced at the sink node, and such variables are updated linearly within each loop iteration. The evaluation would use the loop count multiplying the linear increment or decrement within each iteration to represent the values of such variables.
 - b) This node is within a loop and contains variables that are transitively referenced at the sink node, but such variables are increased or decreased linearly with only with one of the paths in a loop within each iteration. The evaluation would use loop count multiplying the linear increment or decrement to represent the upper or lower boundary of the value ranges.
 - c) This node modifies a variable that is aliased with other variables along this path. The evaluation uses this variable’s modified value as the reaching definition for all the variables within the same alias class. Later, if a variable v ’s latest definition is required and v is in an alias class, the evaluation uses the reaching definition of the alias class as the v ’s definition.

We call the symbolic evaluation with steps (1) to (10) as a restricted form of symbolic evaluation for buffer overflow detection, but in the following, we still use symbolic evaluation to represent it for convenience.

We give an example of the proposed symbolic evaluation using the program in Figure 5-3. The objective is to evaluate the unsatisfiability of the constraint: $user.length \geq (line.length - j)$ from node1 to node7. The symbolic evaluation starts the propagation at node7 and goes backwards along the path. The predicate node6 is not related with the above constraint so it will be skipped. Then node3, 4, 5 are within a loop. So this loop is treated as a basic block according to step (8). The variable j is not updated linearly in this loop. According to step (9), the variable j is first expressed with loop count: $t_1 \leq j \leq t_2$, in which t_1 stands for the loop times in the case that at least one iteration passes through the path (node3, node4, node5); t_2 stands for the loop times in the case that the loop always iterates the path (node3, node4, node3). Afterwards, because t_1 's range is $[0, 0]$ and t_2 's range is $[0, line.length]$, so the j 's value range over the loop would be $[0, line.length]$. After substitution, the constraint becomes: $USERSZ \geq line.length$. Because $USERSZ$ and $line.length$ are two variables mutually independent, we use a single symbol y to represent $USERSZ$ and x to represent $line.length$. According to step (7), the joint expression becomes $\forall y, \forall x, y \geq x$. This expression is submitted to Z3. After the validation of Z3, the joint expression is unsatisfiable. Therefore, the restricted symbolic evaluation finished at node 3. The path from node1 to node7 is vulnerable as it contains an instance of pattern3. This evaluation is light-weighted because only node3 and node7 are involved in the symbolic evaluation and related with constraint solving.

5.2 Buffer Overflow Detection with Code Patterns

In this section, we describe a method to detect buffer overflow vulnerabilities through identifying the three patterns. This method takes given CFGs as input and output the paths that are vulnerable to buffer overflow attack.

The pattern recognition only evaluated selected nodes by using the proposed restricted symbolic evaluation. This helps to reduce the evaluation cost and makes the pattern recognition scalable. Meanwhile, patterns cover different kinds of real situations. The pattern recognition would check the type of the sink nodes first, and then invoke the corresponding pattern recognition procedure.

Moreover, the relationships between paths are utilized to optimise the scalability further. We first introduce **Buffer Overflow Sensitive node** (denoted as *BOS* node) as either of the follows:

- 1) During the pattern recognitions, *BOS* nodes include the predicate nodes that are before a sink node and involves in the restricted symbolic evaluation.
- 2) During the pattern recognitions, *BOS* nodes include the nodes updating variables that are transitively referenced by a sink node.

If a path is identified as vulnerable and supposing the corresponding sub vulnerable path is P , any other path R passing through the same sink node and satisfying the following two relationships immediately are considered as vulnerable ones without any pattern recognitions:

- 1) R contains P .
- 2) R contains the exactly same set of *BOS* nodes in P .

We call the above two relationships as: R shares existing *vulnerable sub routine* with P . In Figure 5-3, the *BOS* nodes are node3 and node7. Any paths share this *vulnerable sub routine* is immediately identified as vulnerable ones without further evaluation.

The proposed method is formally described by an algorithm *detectBufferOverflows*, shown in Figure 5-4. It processes the set of paths in a given CFG, G . Once a path is deemed to match one of the patterns, *detectBufferOverflows* identifies the path containing one buffer overflow vulnerability. The algorithm also identifies the corresponding *vulnerable sub routine* from this path and stores it in the set. The algorithm, *detectBufferOverflows* consists of three sub procedures: *evalPattern1*, *evalPattern2* and *evalPattern3*. Next we shall present three sub procedures first before presenting *detectBufferOverflows* in details.

The function *evalPattern1* possesses two parameters:

- 1) π – the target path in the control flow graph G ,
- 2) S – the sink nodes on π .

And it returns two elements:

- *isPattern1* – a Boolean value to indicate whether the path π matches Pattern1;
- *vulnerableSubRoutine* – the corresponding vulnerable sub path in π .

This function backward traverses π and searches pattern1 for every sink node, only with syntax analysis. During the traversal, once encountered a sink node S_i , a paired element $\langle S_i, N_i \rangle$ will be created and stored as one in a list. The second element N_i is used to store *BOS* nodes related with the pattern1 searching for S_i . Later every time traversing a new node, each paired element $\langle S_i, N_i \rangle$ will be updated until one of them satisfies pattern1 or the entry node of π has been reached. Once *evalPattern1* identifies that π is vulnerable, it picks out the nodes in N_i and returns them as the *vulnerableSubRoutine*.

Input: G : the CFG of a procedure;
Output: $\mathfrak{F}$: the set of paths with buffer overflow vulnerability
Algorithm *detectBufferOverflows*(G)

BEGIN

1. $\mathfrak{F} = \emptyset$;
// the paths with buffer overflow vulnerability
2. $\xi = \emptyset$;
// accumulated *vulnerable sub routines*
3. $l = \text{identifyLoop}(G)$;
4. Traverse Ω using depth-first search,
5. **for** each path π that contains sink node
6. $isVulnerable = \text{false}$;
7. **if** π contains a *vulnerable sub routine* in ξ **then**
8. $isVulnerable = \text{true}$;
9. include π in $\mathfrak{F}$;
10. **continue**;
11. **endIf**
12. **if** S_π is the type sink node of pattern1
13. // inferring vulnerability from Pattern1
14. $(isVulnerable, vulnerableSubRoutine) = \text{evalPattern1}(\pi, S_\pi)$;
15. **if** $isVulnerable = \text{true}$ **then**
16. include π in $\mathfrak{F}$;
17. $\xi = \xi \cup \{ vulnerableSubRoutine \}$;
18. **continue**;
19. **endIf**
20. **endIf**
21. $\psi = \text{aliasAnalysis}(\pi, S_\pi)$;
22. **if** S_π is the type sink node of pattern2
23. // inferring vulnerability from Pattern2
24. $(isVulnerable, vulnerableSubRoutine) = \text{evalPattern2}(\pi, S_\pi, \psi)$;
25. **if** $isVulnerable$ **then**
26. include π in $\mathfrak{F}$;
27. $\xi = \xi \cup \{ vulnerableSubRoutine \}$;
28. **continue**;
29. **endIf**
30. **endIf**
31. **if** S_π is the type sink node of pattern3
32. // inferring vulnerability from Pattern3
33. $(isVulnerable, vulnerableSubRoutine) = \text{evalPattern3}(\pi, S_\pi, \psi)$;
34. **if** $isVulnerable$ **then**
35. include π in $\mathfrak{F}$;
36. $\xi = \xi \cup \{ vulnerableSubRoutine \}$;
37. **continue**;
38. **endIf**
39. **endIf**
40. **endFor**
41. **return** $\mathfrak{F}$
42. **END**

Figure 5-4. Algorithm of pattern recognition

The function *evalPattern2* and *evalPattern3* have the same input and output settings. The settings are similar with *evalPattern1* except containing one more parameter:

- ψ – a set of equivalent classes containing aliased variables along a path.

The value of ψ is processed by the steps (9) and (10) of our restricted symbolic evaluation. Here, the procedure of either *evalPattern2* or *evalPattern3* first uses syntax analysis to find out the sink node and the corresponding *vrn*. The symbolic evaluation then backward traverses π from the sink node, to check whether the *vrn* can guarantee the boundary properties. During the traversal once encountered a sink node, For each sink S_i , along the path, a triple $\langle S_i, \beta_i, N_i \rangle$ will be created and stored in a list. The second element β_i is the used to store the current constraint of pattern2 or pattern3 generated for S_i . The third element N_i is used to store BOS nodes related with the pattern recognition for S_i . Later every time traversing a new node, each triple $\langle S_i, \beta_i, N_i \rangle$ will be updated by substituting variable definitions and joining branch conditions. Once one β_i has been updated into the states that do not contain any local variables, it will be submitted to Z3 for validation. The function *evalPattern2* or *evalPattern3* ends when any of the β_i is proved as FALSE or the entry node of π has been reached. Once π is identified as vulnerable, the sub routine consisting of the nodes in the corresponding N_i is picked and returned as the *vulnerableSubRoutine*.

In *detectBufferOverflows*, initially, $\mathfrak{J}$ is an empty set. Then, a depth-first search is applied to traverse the given control flow graph G . For each path π from the graph entry to any leaf node, if π contains any *vulnerable sub routines* in the set ξ (this is the set of *vulnerable sub routines* in those traversed paths that are accumulated on the fly), it is deemed vulnerable right away without any further processing and included in $\mathfrak{J}$. Otherwise, π is first checked using syntax analysis to pick out the sink node within it. According to the type of the sink node, π will be checked by *evalPattern1*, *evalPattern2* or *evalPattern3* respectively to determine its vulnerability. Once π is identified as vulnerable, the new obtained *vulnerable sub routine* is returned from either *evalPattern1*, *evalPattern2* or *evalPattern3* and will be stored in ξ .

5.3 Empirical Evaluation

5.3.1 Implementation

In order to show the effectiveness of the proposed method, we implemented a prototype system using the CodeSurfer infrastructure [14]. Figure 5-5 shows the architecture of the prototype system. The tool consists of three modules: Program Analyser, CFG Traveller and Vulnerability Detector. In this implementation, Program Analyser receives C/C++ files as inputs. For each input program, program analyser uses CodeSurfer's APIs to build a set of

function calling sequences. Each call sequence consists of a root procedure (functions without any callers in the entire program) and all of those sequentially called sub procedures. An inter CFG is generated for each call sequence and sent to CFG Traveller. CFG Traveller will traverse the entire graph until all paths are checked. Once a path is traversed, it will be sent to Vulnerability Detector. Vulnerability Detector implements the algorithm in Figure 5-4 to perform the pattern recognitions on a given path.

Figure 5-5. Implementation of the proposed method.

5.3.2 Experiment Results

The experiment is carried out on one desktop with Intel Duo E6750 2-core processor, 2.66 GHz, 4 GB memory and Windows XP system. We compare our method with Marple [113] [114], a state-of-the-art path sensitive approach. A detailed review of Marple is given in Chapter 7.2. The comparison focuses on the following two aspects:

- 1) Detection precision: A series of programs are selected to show the percentage of false negative cases and false positive cases.
- 2) Detection scalability: We illustrate the proposed method's efficiency by directly comparing the detection time cost by the proposed method and Marple. We further

explained this by comparing the four items between the two methods: number of nodes involved in symbolic evaluation per path, number of predicate nodes involved in symbolic evaluation, times of calling constraint solver and number of paths involved in symbolic evaluation.

The experiment has into two parts. The first part is carried out on a set of benchmark programs from BugBench [123] and Buffer Overflow Benchmark [181]. The second part is carried out on another three large industrial programs.

In Table 5-1, we show the detailed experimental results of detection accuracy for benchmark systems. Column *System* lists the benchmark programs we tested. Column *KLOC*--thousands lines of codes, shows the size of each benchmark. Column *Detected* shows the number of vulnerability found by the proposed method while Column *Reported* shows the number of vulnerability found by Marple. We have one more manual approval stage to double-check the detection result. The last two columns *FP* and *FN* stand for cases of false positive and false negative that identified in the manual approval stage. As we can see the proposed method detects all the reported buffer overflow vulnerabilities, which means the approach did not scarify any detection precision.

We show the detailed experimental results of detection scalability for benchmark systems in Table 5-2. Though the benchmark systems are relatively small, the proposed method has also shown its efficiency. We compare the detection time on each benchmark in Column *time*. The proposed method respectively saved 26.47% amount of processing time per path in average than Marple. The time saving comes from four reasons, illustrated by the other four Columns:

- 1) *#NodeEva*: it stands for the average number of nodes evaluated per path. The proposed method saved 37.82% in average in this comparison.
- 2) *#PreNodeEva/path*: it stands for the average number of predicate nodes evaluated per path. The proposed method saved 17.96% in average in this comparison.
- 3) *#call Solver*: it stands for the average times calling constraint solver per path. The proposed method saved 73.68% in average in this comparison.
- 4) *#Path*: it stands for the total number of paths evaluated per benchmark system. The proposed method saved 58.60% in average in this comparison.

In Table 5-3 and Table 5-4 we compare the detection accuracy and scalability in large applications. In Marple, there is no abstraction mechanism to handle loops. Therefore, we explicitly set Marple to iterate up to 10 times only for each loop.

In Table 5-3, label PM and MA stands for the proposed method and Marple. The column *Detected* stands for the number of vulnerable paths automatically detected by the proposed method or Marple. The column FP^* stands for cases of false positive. The column FN^* stands for cases of false negative. We found out FP^* and FN^* by double examined the detected paths against CodeSonar [13], another static analysis tool, and manual work. The proposed method abstracts the relationships between loops and sink node while Marple does not work well for programs containing complex loops. Marple would cause a large amount of false negative cases while the proposed method shows its high accuracy in large programs by identifying these cases correctly.

In Table 5-4, we followed the same setting with Table 5-2 to compare the scalability. We found in average the proposed method saved 81.02%, 96.45%, 96.62%, 97.49%, and 81.89% for the 5 groups of comparisons, respectively. The proposed method performs much better than Marple in large applications are mainly because:

- 1) The proposed method first uses syntax analysis to filter nodes and paths and only send selected nodes and paths for symbolic evaluation. Many nodes and predicates nodes are processed with light-weighted syntax analysis while theses nodes are processed with a heavy symbolic evaluation by Marple.
- 2) The proposed method utilizes existing vulnerable paths helping to identify other vulnerable paths sharing the same *BOS* nodes.
- 3) The proposed method abstracts many loops as basic blocks while Marple has to iterate the loop for fixed times.
- 4) When the program sizes grows, Marple requires much more times calling constraint solver, which causing the solver having more times stuck and hang.

Table 5-1. Accuracy of Proposed Method on Benchmarks

System	KLOC	Detected	Reported	FP	FN
Polymorph-0.4.0	1.7	15	15	0	0
Ncompress-4.2.4	2.0	38	38	0	0
Gzip-1.2.4	8.2	38	38	0	0
Bc-1.06	17.7	245	245	0	0
Wu- ftdp-2.6.2	0.4	13	13	0	0
Sendmail-8.7.5	0.7	21	21	0	0
BIND	1.3	48	48	0	0
Total	32	418	418	0	0

Table 5-2. Scalability Comparison between Proposed Method and Marple on Benchmarks

System	Time(s)		#NodeEva/path		#PreNodeEva/path		#call Solver		#Path	
	PM	MA	PM	MA	PM	MA	PM	MA	PM	MA
Polymorph0.4.0	0.79	1.27	2.16	3.22	1.51	1.89	1.5	2.92	83	321
Ncompress4.2.4	0.99	1.41	2.87	5.13	1.03	1.77	1.6	3.97	78	341
Gzip-1.2.4	19.4	24.33	3.05	4.64	2.04	2.53	1.1	4.25	373	517
Bc-1.06	148.81	203.36	4.91	8.57	3.10	1.67	1.4	7.15	197	663
Wu-ftp2.6.2	1.21	2.43	5.68	8.19	1.92	3.18	1.2	6.87	318	459
Sendmail-8.7.5	0.65	1.07	3.92	7.32	2.01	2.54	1.5	6.50	140	487
BIND	0.70	0.82	5.33	7.83	3.37	4.68	1.8	6.72	56	219
Total	172.55	234.69	27.92	44.90	14.98	18.26	10.10	38.38	1245	3007

Table 5-3. Accuracy of Proposed Method on Industrial Applications

System	KLOC	PM			MA		
		Detected	FP*	FN*	Detected	FP*	FN*
RounterCore	137.15	319	12	2	153	3	159
PathFinder	104.23	107	8	4	76	1	28
RFIDScan	219.36	422	17	1	178	2	230
Total	460.74	848	37	7	407	6	417

Table 5-4. Scalability between Proposed Method and Marple on Industrial Applications

System	Time(min)		#NodeEva/path		#PreNodeEva/path		#call Solver		#Path	
	PM	MA	PM	MA	PM	MA	PM	MA	PM	MA
RounterCore	15.31	47.92	4.89	237.01	1.05	53.29	1.75	83	1793	4508
PathFinder	5.67	29.84	6.20	62.54	2.71	28.60	0.93	14	691	2940
RFIDScan	11.43	93	10.79	318.39	6.98	235.77	1.24	59	1038	12004
Total	32.41	170.76	21.88	617.94	10.74	317.66	3.92	156	3522	19452

5.4 Limitation and discussion

However we also observed that the proposed method also produced some false negatives. This is because the industrial applications contain many third-party objects. Additionally, we missed considering some user-defined operations that perform the same tasks with the standard C functions like *strcpy*.

We also observe that the proposed method leads to a number of false positive cases. We investigated the reasons and found the identification procedure will consider a path vulnerable when it identifies a *pattern1* instance on this path. However, we found that though a small number of paths contain *pattern1*, but the sink nodes are protected by operations that manipulate inputs by removing escape characters. These paths are safe actually.

5.5 Conclusion

The goal of this chapter was to improve the current state of the art in buffer overflow detection by making the detection in real systems more efficient and scalable. We have identified and defined three patterns in the CFGs of real systems that lead to paths with vulnerabilities:

pattern of vulnerable syntax usage, pattern of vulnerable element accessing and pattern of vulnerable bulk moving. We have proposed a method to detect such vulnerable paths by directly traversing the given CFG in a scalable way.

The proposed method focuses on scalability while not sacrificing the precision of detection by making limited use of the theorem prover Z3 to selected paths and branch predicates. This selection is made according to the patterns that characterize most of the paths, which contain buffer overflow vulnerabilities, in real systems. Hence, it is also a form of path sensitive analysis. As opposed to other works, our main contribution is to improve the state of the art as follows:

- 1) We modelled the common buffer overflow vulnerability into three codes patterns.
- 2) We proposed an approach to recognize these patterns. This approach is scalable and accurate as it consists of syntax analysis and a light-weighted restricted symbolic evaluation.
- 3) We presented two relationships between paths, helping to enhance the scalability further.
- 4) The extensive experiments on the selected C/C++ systems proved the efficiency of the proposed approach.

Chapter 6

AUTOMATIC REMOVAL OF BUFFER OVERFLOW VULNERABILITIES IN C/C++ PROGRAMS

To mitigate the threats of buffer overflow attacks, a number of approaches have been proposed. Existing approaches and tools focus mainly on three directions [176]:

- 1) the prevention of run-time buffer overflow attacks by creating a run-time environment, like a sandbox so that tainted input could not directly affect certain key memory locations;
- 2) the detection of buffer overflows in programs by applying static checking techniques to analyse source code;
- 3) the transformation of the original program by adding additional verification code or external annotations.

However, none of the existing methods fully resolved the threats of buffer overflow attacks. For approaches on the first direction, as modern programs are becoming more complex, it is difficult to develop a universal run-time defence [176]. For approaches on the second direction, even if buffer overflow vulnerabilities are detected, the vulnerable programs are still being used until new patches are released. For the third direction, though it is well-motivated to add extra validation to guard critical variables and operations, the existing approaches will add considerable runtime overhead. For example, a recent novel approach that adds extra bounds checking for every pointer may increase the runtime overhead by 80% on average [132].

In this chapter, we propose a framework along the third direction, to improve the state-of-the-art. This framework iteratively detects and removes buffer overflows in a path-sensitive manner, until all the detected vulnerabilities are eliminated. Unlike the related methods [54, 59, 90, 132], the proposed framework only patches identified code segments, thus it can eliminate buffer overflows while keeping a minimum runtime overhead.

We have evaluated the proposed approach on a set of benchmarks and three industry C/C++ applications. The results show that the proposed approach is effective. First it can remove all the detected buffer overflow vulnerabilities in the studied subjects. Second we also compare it with other methods that focus on buffer overflow removal. On average, it removes 58.28% more vulnerabilities than methods that apply a straight-forward “search & replace” strategy; it inserted 72.06% fewer predicates than a customized bounds checker.

6.1 The Proposed Approach

In this section, we propose **Automatic Buffer Overflow Repairing (ABOR)**, a framework which integrates the detection and removal mechanisms to resolve the buffer overflow vulnerabilities in a given program automatically. Figure 6-1 demonstrates the overall structure of the framework. ABOR consists of two modules: vulnerability detection and vulnerability removal. ABOR works in an iterative way: once the vulnerability detection module captures a vulnerable code segment, the segment is fed to the removal module; the fixed segment will be patched back to the original program. ABOR repeats the above procedure until the program is buffer-overflow-free. In this section, we introduce the two modules of ABOR in detail.

Figure 6-1. Overview of ABOR

6.1.1 Preliminary for ABOR

We first review some basic definitions of static analysis [29, 155]. A control flow graph (CFG) for a procedure is a graph that visually presents the control flow among program statements. A path is one single trace of executing a sequential of program statements. A variable in a program is called an input variable if it is not defined in the program solely from constants and variables. An abstract syntax tree (AST) is a tree structure that represents the abstract syntax of source code written in a particular programming language. We call a program operation that may cause potential buffer overflows as a buffer overflow sensitive sink (bo-sensitive sink). For convenience, in this chapter, we shorten the name “bo-sensitive sink” to sink. The node that contains a sink is called a sink node.

There are many methods to protect sinks from being exploited [110, 115-117, 133]. A general way is to add constraints to protect sink from being exploited [176]. This could be expressed as the following rule:

General Rule: To prevent buffer overflow, before any execution of a sink, there shall be a protection constraint to ensure certain properties.

After a careful review, we collected a list of common sinks and the corresponding protection constraints. The collected information is to concrete the above general rule with four detailed rules. Each rule represents a type of common buffer overflow cases in C/C++.

Rule#1: A statement is a bo-sensitive sink when it defines or updates a destination buffer with an input from either (1) a C stream input function which is declared in <stdio.h>, or (2) a C++ input function inherited from the base class istream. The protection constraint shall ensure the length of the destination buffer is longer than the input data.

Rule#2: A statement that copies/moves the content of a block of memory to another block of memory is a bo-sensitive sink. The protection constraint shall ensure that the copied/moved data is not larger than the length of destination memory block.

Rule#3: A statement that defines an array element is a bo-sensitive sink. If the array element is indexed by an integer variable *v*, the protection constraint shall validate that *v* is within the range between 0 and the array size. If the array element is indexed by a pointer *p*, then the protection constraint shall ensure that the address of *p* is within the array boundary.

Rule#4: A statement is a bo-sensitive sink when it calls a C stream output function when (1) this function is declared in <stdio.h>; or (2) this function contains a format string that mismatches its corresponding output data; or (3) the function's output is data dependent on its parameters [116]. The protection constraint shall ensure that:

- 1) the output data should not contain any string derived from the prototype [6]:
`%[flags][width][.precision][length]specifier;`
- 2) or all the character “%” in the output data has been encoded in a backslash escape style, as “\%”.

We use the above prevention rules as a guideline to construct the buffer overflow detection and removal modules of ABOR. Each prevention rule covers a series of cases in C/C++. In order to specify these cases clearly, we use metadata to describe them exhaustively at AST level. These metadata are maintained in Table 6-1. Each row stands for a buffer overflow case. The column *Sink* lists the AST structure of sinks. The column *Protection Constraint* specifies the AST structure of the constraint which could prevent the sinks being exploited. Additionally, in order to concrete the protection constraints ABOR needs to substitute in constants, local variables, expressions. Table 6-1 empirically collects the most common types of buffer overflow vulnerabilities. It is sound, but theoretically it is not complete. But in practice, users of ABOR can also extend this table for better coverage.

Table 6-1. Detail of ABOR rule table

#Rule	Sink	Protection Constraint	Required Intermediate Variable
1	gets(dst)	$\text{dst_length} \geq \text{SIZE_MAX}^1$	dst_length
	fgets(dst, n, stream)	$\text{dst_length} \geq n$	//The destination buffer
	fread(dst, n, count)	$\text{dst_length} \geq n * \text{count}$	dst's length, in terms of
	Operator '>>'	$\text{dst_length} \geq \text{SIZE_MAX}$	bytes.
	Operator '>>' with the width function setw(t)	$\text{dst_length} \geq \text{strlen}(\text{str})$	
	getline(dst, str, ..)		
	read(dst, str)		
	readsomewhat(dst, str)		
	fscanf(..., *format, dst,...)	If the input format specifier '*format' contains input width, then the constraint is: $\text{dst_length} \geq \text{width};$ Else the constraint is: $\text{dst_length} \geq \text{SIZE_MAX}$	
	scanf(*format, dst,...)		
2	sscanf(src,*format, dst,..)	If the input format specifier '*format' contains input width, then the constraint is: $\text{dst_length} \geq \text{width};$ Else the constraint is: $\text{dst_length} \geq \text{src.length}$	
	memcpy(dst, src, t)	$\text{dst_length} \geq t;$	dst_length,
	memmove(dst, src,t)	$\text{src_length} \geq t;$	//The destination buffer
	strncpy(dst,src, t)	$\text{dst_length} \geq t;$ $\text{src_length} \geq t;$ $\text{src}[\text{strlen}(\text{src})+1] == '\0'$	dst's length, in terms of
	strcpy(dst, src)	$\text{dst_length} \geq \text{src_length};$ $\text{src}[\text{strlen}(\text{src})+1] == '\0';$	src_length
	memset(dst, v, n)	$\text{dst_length} \geq n;$	//The source buffer src's
	strcat(dst,src)	$\text{src}[\text{strlen}(\text{src})+1] == '\0';$ $\text{src}[\text{strlen}(\text{dst})+1] == '\0';$ $\text{dst_length} \geq \text{strlen}(\text{dst}) + \text{strlen}(\text{src});$	length, in terms of bytes.
	strncat(dst,src, n)	$\text{src}[\text{strlen}(\text{src})+1] == '\0';$ $\text{src}[\text{strlen}(\text{dst})+1] == '\0';$ $\text{dst_length} \geq n$ $\text{strlen}(\text{dst}) + \text{strlen}(\text{src});$	
	realloc (src, newlen)	$\text{src_length} * \text{sizeof}(\text{src}[0]) \leq \text{newlen};$	
	T *t=new (&dst) T()	$\text{sizeof}(T) \leq \text{dst_length};$ //placement new	
3	array[i]	$\text{sizeof}(\text{array})/\text{sizeof}(\text{array}[0]) \geq i$	
	*p	$p_index \geq 0;$ $p_index \leq p_length / p[0]$	p_index /*an integer value, initialized as 0, standing for the corresponding integer index for the pointer p, in terms of bytes; p_length /*the value range of the pointer, in terms of bytes*/
4	The family of output	/*Define: */	

¹ SIZE_MAX stands for the max value of unsigned long

	<p>functions specified by [20], includes: fprintf, printf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf.</p> <p>And the output data is data dependent on input.</p>	<pre>std::tr1::cmatch res; std::string regex_match = the output data; std::tr1::regex pattern(the regular expression derived from the prototype of %[flags][width][.precision][length]specifier); // See the reference [20] /*Constraint: */ regex_search(regex_match.begin(), regex_match.end(), pattern) ==FALSE; OR std::tr1::regex_search(regex_match.c_str() , res, rx); for(each found sub string res[i]){ (*res[i]--) =='\'; }</pre>	
--	--	--	--

Table 6-1 also contains two more critical data structures: (1) buffer length and (2) buffer index. In C/C++, there is no universal way to retrieve the above data easily. To solve this, ABOR creates intermediate variables to represent them. In Table 6-1, the last column *Required Intermediate Variables* of records such kinds of intermediate variables for each constraint.

6.1.2 Buffer Overflow Detection

Among the existing detection methods, approaches working in a path sensitive manner offer higher precision because they target at modelling the runtime behaviour for each execution path:

- 1) For each path, path sensitive approaches eventually generate a path constraint to reflect the relationship between the external input and target buffer.
- 2) The path's vulnerability is verified through validating the generated path constraint.

In the current implementation of ABOR, we modified a recent buffer overflow detection method called Marple [113] [114] and integrated it into ABOR. Marple is a state-of-the-art path sensitive approach for vulnerability detection. A detailed review of Marple is given in Chapter 7.2.

Marple maintained a lookup table to store the common bo-sensitive sinks' syntax structure. For each recognized bo-sensitive sink node and the path passing through the sink node, Marple generates an initial constraint, called query and backward propagates the query along the path. The constraint will be updated through symbolic execution when encountering nodes that could affect the data flow information related to the constraint. Once the constraint is updated, Marple tries to validate it by invoking its theorem prover. If it is proved the constraint is unsatisfiable, Marple concludes this path buffer-overflow-vulnerable. Figure 6-2 demonstrates how ABOR integrates Marple:

- 1) ABOR updates Marple's lookup table with Figure 6-1. We enforce Marple to search for the syntax structures listed in the column *Sink* of Figure 6-1. Additionally, Marple will raise query based on the column *Protection Constraint*.
- 2) ABOR uses a depth-first search to traverse a given procedure's control flow graph: each branch will be traversed once. If a bo-sensitive sink is found, the segment starting from the sink back to the procedure entry will be constructed to be a set of paths and each path will be fed to Marple for processing.
- 3) ABOR updates Marple's constraint solver with Z3 [130], a latest SMT solver from Microsoft with stronger solvability.
- 4) If one path is identified vulnerable, ABOR records the sub-path that causes the vulnerability or infeasibility [113]. Later, paths containing any of such sub-paths will not be examined.
- 5) We follow the way Marple handles loop structures. Marple treats a loop structure as a unit and tries to compute each loop's impact on the propagating constraint, if and only if such impact is linear.

We illustrate the above procedure with an example shown by the left column of Table 6-5 is a sink node, therefore, Marple raises a constraint as $pvpbuf_length \geq req_bytes_length$, and it propagates backward along the paths that pass through *s1* and tries to evaluate the constraint. For example:

- 1) Along the path (*n1, n2, n3, n4, n5, n6, n7, n8, n14, n15, n16, s1*), the constraint is updated at *n8* and *n5* and becomes $max_size \geq 1024$ at node *n5*. In the context, *max_size* is the macro of *1024*. So the constraint becomes $1024 \geq 1024$ and is evaluated as valid. Therefore, this path is identified as being safe.
- 2) Along the path (*n1, n2, n3, n4, n5, n6, n7, n9, n10, n11, n12, n9, n13, n14, n15, n16, s1*), the constraint is updated at node *n13* and becomes $c2 \geq 1024$. The variable *c2* is affected by the loop [*n9, n10, n11, n12, n9*]. The variable *c2* is used in the loop, and it is data dependent on the input-*ADDRSIZE*. It is able to find a counterexample to violate the constraint $c2 \geq 1024$. So this path is identified as being vulnerable. The method introduced in Section 6.1.3 will be used to remove this vulnerability.

Figure 6-2. Vulnerability detection in ABOR

6.1.3 Buffer Overflow Removal

The removal module takes an identified vulnerable path, analyses the sink node's syntax tree and picks the corresponding constraint to protect the sink. The main challenge is to transform the selected constraint into valid C++ code.

6.1.3.1 Intermediate variable

ABOR propagates backward along the given path to enable the intermediate variables simulating their semantics. It maintains Table 6-2 to assist this propagation. In Table 6-2, the column *Syntax Pattern* stands for the AST ABOR searches for during the propagation. The column *Update Operation* stands for how ABOR updates the corresponding intermediate variables.

Table 6-2. Update Intermediate variable during propagation

Syntax Pattern	Update Operation
Buffer definition	
<ul style="list-style-type: none"> • <code>buffer = new wildcard T [n];</code> • <code>buffer = calloc(n, wildcard²);</code> • <code>T buffer[n] = { wildcard };</code> • <code>T buffer[n];</code> 	<code>buffer_length = n*sizeof(T);</code> <code>buffer_index=0;</code>
<ul style="list-style-type: none"> • <code>T *buffer = new T wildcard;</code> 	<code>buffer_length = sizeof(T);</code> <code>buffer_index=0;</code>
<ul style="list-style-type: none"> • <code>T*buffer =new T(s);</code> • <code>buffer = malloc(s, wildcard);</code> • <code>buffer = realloc(wildcard, s);</code> 	<code>buffer_length = s ;</code> <code>buffer_index=0;</code>
Buffer referencing	
<ul style="list-style-type: none"> • <code>T * p=buffer;</code> 	<code>p_length=buffer_length;</code> <code>p_index=buffer_index;</code>
Array index subscripting	
<ul style="list-style-type: none"> • <code>buffer[i] = wildcard</code> 	<code>buffer_index=i;</code>
Pointer arithmetic	
<ul style="list-style-type: none"> • <code>p++;</code> 	<code>p_index ++;</code>
<ul style="list-style-type: none"> • <code>p--;</code> 	<code>p_index--;</code>
<ul style="list-style-type: none"> • <code>p=p+n;</code> 	<code>p_index = p_index +n;</code>
<ul style="list-style-type: none"> • <code>p=p-n;</code> 	<code>p_index = p_index - n;</code>
Free memory	
<ul style="list-style-type: none"> • <code>free (p);</code> • <code>delete [] p;</code> • <code>delete p;</code> 	<code>p_length=0;</code> <code>p_index=NULL;</code>

6.1.3.2 Program transformation for vulnerability removal

The algorithm *removeVul* shown in Table 6-3 shows the workflow of ABOR's removal module. This algorithm takes an identified vulnerable path *pth* and its control flow graph *G* as parameters. It inserts defensive code into *G* to remove the vulnerability. The defensive code includes statements to update intermediate variables and statements to protect the sink.

Given an identified vulnerable path and the sink node, *removeVul* first retrieves the metadata from Table 6-1 based on the sink node's syntax. The metadata is returned as a pair $\langle \delta, \{v\} \rangle$, where δ stands for the protection constraint and $\{v\}$ stands for the required intermediate variables. *removeVul* propagates along the path backward from the sink node to the entry node. The loop in *removeVul* from line4 to line12 compares each traversed node's AST with the syntax patterns listed in Table 6-2. If the syntaxes match, *removeVul* inserts a node into the control flow graph. These inserted nodes contain the corresponding operations in Table 6-2. In the runtime, they will update the intermediate variables to simulate the target buffer length and index. After this, *removeVul* concretes the protection constraint by substituting required local variables, expressions, constants and intermediate variables. The protection constraint is used to construct a predicate node. This predicate node wraps the sink

² *wildcard* here stands for arbitrary characters from zero to many.

node to provide protections. At last, *removeVul* will convert the modified control flow graph back to source code. Therefore, the vulnerability caused by the sink has been removed by ABOR. This program transformation is illustrated by Figure6-2.

In Table 6-4, we apply *removeVul* on four pieces of code fragments. Each code piece is for one prevention rule listed in Table 6-1. After executing *removeVul*, the vulnerable code fragments are repaired as the sink nodes are protected with constraints.

Table 6-3. Vulnerability removal in ABOR

Input:	<i>G</i> : the CFG of a procedure; <i>pth</i> : the identified vulnerable path <i>sink</i> : the sink node
Global Variables:	δ : based on Table 6-1. Detail of ABOR rule table , δ is the protection constraint for the sink $\{v\}$: based on Table 6-1. Detail of ABOR rule table , $\{v\}$ is the set of required intermediate variables
Output:	<i>G'</i> : the CFG with the inserted defensive code <i>SrcFile'</i> : the converted program

<p>Algorithm <i>removeVul</i> (<i>G</i> , <i>pth</i> , <i>s</i>)</p> <p>begin</p> <ol style="list-style-type: none"> 1. $\delta = \text{NULL}$; $\{v\} = \emptyset$; $G' = G$; 2. CFGNode <i>cur_node</i> = NULL; //the current traversed CFGNode 3. $\langle \delta, \{v\} \rangle = \text{LookupT1}(\textit{sink})$; //this sub-procedure lookups in Table 6-1 and gets the metadata 4. for (<i>cur_node</i> := From <i>sink</i> To <i>pth</i>'s entry node) do // update intermediate variables 5. for each <i>v</i> in $\{v\}$ do 6. Insert a CFGNode into <i>G'</i> as its Entry Node's immediate post-dominator, to declare <i>v</i> and initialize $v=0$; 7. if ($\text{LookupT2}(\textit{cur_node}, v) == \text{TRUE}$) then /*this sub-procedure lookups in Table 6-2 to check whether the current CFGNode contains AST matching the Syntax Patterns listed in Table 6-2*/ 8. Insert a CFGNode into <i>G'</i> as <i>cur_node</i>'s immediate post- dominator, to perform the corresponding update operation listed in Table 6-2; 9. endIf 10. endFor 11. endFor 12. Concrete δ with required local variables, expressions, constants and intermediate variables in $\{v\}$; 13. Use δ as condition to construct a predicate and insert this predicate node into <i>G'</i> as <i>sink</i>'s immediate dominator's immediate post-dominator; 14. Place <i>sink</i> and the rest part of <i>G'</i> on the predicate's TRUE branch; 15. Add an exception-handling node on the predicate's FALSE branch and link this FALSE branch to <i>G</i>'s exit node. 16. $G' \rightarrow \text{SrcFile'}$ // convert <i>G'</i> back to source code <p>end</p>	
---	--

Figure 6-3. Program Transformation

Table 6-4. removeVul for different prevention rules

Rule	Vulnerable program	Repaired program
1	<pre>int dst[len]; read(dst, 32);</pre>	<pre>int tempL_dst=0; int dst[len]; tempL_dst=len; if(tempL_dst>=32) read(dst, 32); else Log("Buffer overflow");</pre>
2	<pre>void func(int * src) { int *dst = new int[256]; strcpy(dst, src); }</pre>	<pre>void func(int * src){ int tempL_src=SIZE_MAX; int tempL_dst=0; int *dst = new int[256]; templeL_dst =256; if(templeL_dst>= tempL_src) strcpy(dst, src); else Log("Buffer overflow");}</pre>
3	<pre>char * dst=new char[n]; char * p=dst; for(int i>0;i<size; i++) { if(...) p++; cout<<*p; }</pre>	<pre>int tempL_dst=0; int index_dst=0; char * dst=new char[n]; tempL_dst=n; char * p=dst; for(int i>0;i<size; i++){ if(...) { p++; index_dst ++; } if(index_dst < temple_dst) cout<<*p; else Log("Buffer overflow");</pre>
4	<pre>void func(char * data) { printf(data); }</pre>	<pre>void func(char * data){ string pattern=....; /* pattern here is a regular expression stated in Table 6-1*/ /*validation node*/ if(std::regex_match (data, std::regex(pattern) ==false) printf(data); else Log("Buffer overflow");</pre>

6.1.4 An example of ABOR

Table 6-5. An example of applying AROR

Sample path	Sink	Prevention Rule	Abstract Constraint
(n1,n2,n3,n4,n5,n6,n7,n9-n13,n14,n15,n16,s1)	strcpy(pvpbuf, req_bytes)	Rule 1	length(pvpbuf) ≥ length(req_bytes)
Vulnerable Program	Repaired Program		
<pre> #define min_size 10; #define max_size 1024; /*addr is an existing array with the size of ADDRSIZE, n2 and n16 are two basic blocks that are not related to pvpbuf and req_bytes*/ int _encode (bool bslashmode, char *addr, int ADDRSIZE){ /*n1*/ do{ /*n2*/ BLOCK1; /*n3*/ }while(...); /*n4*/ int c1=0; int c2=0; /*n5*/ int PSBUFSIZE= max_size; /*n6*/ int *pvpbuf; /*n7*/ if (bslashmode == TRUE){ /*n8*/ pvpbuf = new int [PSBUFSIZE]; } else{ /*n9*/ while(c1<ADDRSIZE){ /*n10*/ if(addr[c1]!='#') /*n11*/ c2++; /*n12*/ c1++; } /*n13*/ pvpbuf = new int [c2]; } /*n14*/ char req_bytes[1024]; /*n15*/ scanf("%1024s", req_bytes); /*n16*/ BLOCK2; /*s1*/ strcpy(pvpbuf,req_bytes); } //END _encode </pre>	<pre> #define min_size 10; #define max_size 1024; /*addr is an existing array with the size of ADDRSIZE, n2 and n16 are two basic blocks that are not related to pvpbuf and req_bytes*/ int _encode (bool bslashmode, char *addr, int ADDRSIZE){ /*p1*/ int pvpbuf_temp0_size=0; int req_bytes0_size=0; /*n1*/ do{ /*n2*/ BLOCK1; /*n3*/ }while(...); /*n4*/ int c1=0; int c2=0; /*n5*/ int PSBUFSIZE= max_size; /*n6*/ int *pvpbuf; /*n7*/ if (bslashmode == TRUE){ /*n8*/ pvpbuf = new int [PSBUFSIZE]; /*p2*/ pvpbuf_temp0_size=PSBUFSIZE; } else{ /*n9*/ while(c1<ADDRSIZE){ /*n10*/ if(addr[c1]!='#') /*n11*/ c2++; /*n12*/ c1++; } /*n13*/ pvpbuf = new int [c2]; /*p3*/ pvpbuf_temp0_size=c2; } /*n14*/ char req_bytes[1024]; /*p4*/ req_bytes_temp0_size=1024; /*n15*/ scanf("%1024s", req_bytes); /*n16*/ BLOCK2; /*p5*/ if(pvpbuf_temp0_size >= req_bytes_temp0_size) /*s1*/ strcpy(pvpbuf,req_bytes); else /*p6*/ cerr<<"Attempt to cause buffer overflow reject"; } //END _encode </pre>		

We present a full example of using ABOR, shown in Table 6-5. The vulnerable program is listed in the left side column. The ABOR's buffer overflow detection module finds that the path (n1, n2, n3, n4, n5, n6, n7, n9-n13, n14, n15, n16, s1) is vulnerable. The path will be passed to *removeVul*. The algorithm traverses the sink node s1's AST and determines that the first prevention rule shall be applied. The constraint is to validate that the length of *pvpbuf* is larger than or equal to the length of *req_bytes*. ABOR inserts one node p1 into the CFG and creates two intermediate variables with initial value of 0: *pvpbuf_temp0_size* for the length of *pvpbuf* and *req_bytes_temp0_size* for the length of *req_bytes*. ABOR inserts another three nodes p2, p3, and p4 into CFG, to manipulate the two temporary variables to track the lengths of *pvpbuf* and *req_bytes*. At last, ABOR constructs the protection constraint as

$pvplib_temp0_size \geq req_bytes_temp0_size$ and transforms the original program by wrapping statement s1 with statements p5, and p6. In the end, ABOR converts the modified CFG back to source code, which is listed in the right side column of Table 6-5. Therefore, the vulnerability has been removed.

6.2 Evaluation

6.2.1 Experiment Design

We have implemented ABOR's prototype based on the CodeSurfer infrastructure [14]. The prototype has two parts: Program Analyser and ABOR. The Program Analyser receives C/C++ programs as input and utilizes CodeSurfer to build an inline inter-procedural CFG. This CFG is then sent to the ABOR for the vulnerability detection and removal.

Seven benchmark programs from Buffer Overflow Benchmark [181] and BugBench [123] (namely Polymorph-0.4.0, Ncompress-4.2.4, Gzip-1.2.4, Bc-1.06, Wu-ftdp-2.6.2, Sendmail-8.7.5, BIND) are selected for the evaluation. To further evaluate the effectiveness of the proposed approach on real-world programs, we also collect three C/C++ programs from industry (namely RounterCore, PathFinder, and RFIDScan).

For benchmark programs, as the buffer overflow vulnerabilities have already been reported in existing work [123, 181], the experiments are to show whether our tool ABOR can remove all the reported vulnerability. For the industry programs, we first detect the buffer overflow vulnerabilities and then try to remove them. The results are manually investigated to check their correctness.

The experiments are carried out on a desktop computer with Intel Duo E6750 2-core processor, 2.66 GHz, 4 GB memory and Windows XP system.

6.2.2 Experimental Results

6.2.2.1 System Performance

Removal accuracy: For benchmark systems, the experiment objective is to show whether ABOR can remove all the vulnerabilities reported in previous work [123, 181].

A false negative case occurs if we manually find that the proposed method failed to remove one buffer overflow vulnerability. A false positive case occurs if we manually find the proposed method patches a piece of code that is actually buffer overflow free. We calculate error rate of the proposed method by dividing the total number of vulnerabilities by the sum of the number of false positives and false negatives.

The experimental results are shown in Table 6-6. The column stands for thousands of lines

of code. The column *#Reported* records the real reported number of vulnerabilities while column *#Repaired* records the number of vulnerabilities removed by ABOR. The columns *#FP* and *#FN* stand for the numbers of false positives and false negatives respectively. For all the systems, the total number of reported buffer overflow vulnerabilities is 418.

For the benchmark programs, Table 6-6 shows that ABOR can automatically remove all the vulnerabilities in the given benchmarks.

For industry programs, the results are shown in Table 6-7. The columns *#Repaired*, *#FN* and *#FP* have the same meaning as those in Table 6-6. The column *#Detected* stands for the number of detected vulnerabilities. The column *#Manual* stands for the number of vulnerabilities discovered from manual investigations. The manual investigation double checked the detection result and also manually analysed the paths that cannot be processed. As shown in Table 6-7, our approach is able to detect 608 buffer overflow vulnerabilities and can successfully remove all of them. The results confirm the effectiveness of the proposed approach in removing buffer overflow vulnerabilities. However, due to implementation limitations, currently ABOR is not able to detect all buffer overflow vulnerabilities. On average, the error rate of our proposed method is 20.53%, which consists of 19.80% false negative cases and 0.73% false positive cases. The details of the error cases are discussed in section 6.2.2.3

Time Cost: We measured the time performance of the proposed approach on both benchmarks and industry programs. Table 6-8 records the time spent on processing each program individually. The proposed approach is able to process complex programs including industry programs. The time cost over the entire 10 systems is 4480 second, which is nearly 75 minutes. It is also discovered that a large amount of time is spent on vulnerability detection, which is 77.77% of the total time. The vulnerability removal process is relatively lightweight, which costs only 22.23% of the total time.

Table 6-6. Vulnerabilities removed by ABOR in benchmarks

System	KLOC	#Reported	#Repaired	#FP	#FN
Polymorph-0.4.0	1.7	15	15	0	0
Ncompress-4.2.4	2.0	38	38	0	0
Gzip-1.2.4	8.2	38	38	0	0
Bc-1.06	17.7	245	245	0	0
Wu- ftpd-2.6.2	0.4	13	13	0	0
Sendmail-8.7.5	0.7	21	21	0	0
BIND	1.3	48	48	0	0
Total	32.0	418	418	0	0

Table 6-7. Vulnerabilities removed by ABOR in industry programs

System	KLOC	#Detected	#Repaired	#Manual	#FP	#FN	ErrorRate(%)
RounterCore	137.15	217	217	309	3	41	14.23
PathFinder	104.23	79	79	103	1	25	25.24
RFIDScan	219.36	312	312	406	2	96	24.13
Total	460.74	608	608	818	6	162	20.53

Table 6-8. The time performance of ABOR

System	Total Time (ms)	Detection Time		Removal Time	
		Time(ms)	%	Time (ms)	%
Polymorph	95.25	81.91	85.99	13.34	14.01
Ncompress	214.32	160.81	75.03	53.51	24.98
Gzip	3698.16	1309.45	35.41	2388.71	64.59
Bc	149469.6	132026.9	88.33	17442.66	11.67
Wu- ftpd	221.13	185.33	83.81	35.80	16.19
Sendmail	134.82	103.76	76.96	31.06	23.04
BIND	187.13	142.55	76.18	44.58	23.82
RounterCore	2446656.17	1781649.13	72.82	665007.07	27.18
PathFinder	654987.43	564359.17	86.16	90628.22	13.84
RFIDScan	1224366.67	1003880.72	81.99	220485.98	18.01
Total	4480227.48	3484075.91	77.77	996151.55	22.23

6.2.2.2 Comparison

There are two types of commonly used removal methods, which are “search & replace” and “bounds checker”. The first category of methods replaces common vulnerable C string functions with safe versions. As a large number of buffer overflows are caused by C string functions, this category of methods can high good effect. Additionally, they are easy for implementation, so they are widely used in practice [129].

ABOR is the method that patches detected vulnerable code segment. So it guarantees the removal precision while it can keep a low runtime overhead. Using the same benchmarks and industry programs, we compare ABOR with the other two important categories of removal methods.

First, we compare ABOR with “search & replace” methods. As the fast development of attacking techniques based on buffer overflows [176], the “search & replace” methods will miss many buffer overflows in real code. In Figure 6-4, we compare ABOR with a recent “search & replace” implementation from Hafiz and Johnson [90]. This figure represents the comparison on the number of removed buffer overflows. It contains a pair of bars for each test case, larger bars are better as they represent the higher removal precision. On average, ABOR can remove 59.75% more vulnerabilities than the implemented “search & replace” method.

Second, we compare ABOR with “bounds checker” methods. This type of methods chases high precision by inserting effective validation before every memory access. In practice, they

are usually used by mission-critical systems[132, 176]. However they usually bring in high runtime overhead as a number of inserted validations are redundant. At the current stage, we implemented a customized bounds checker following a novel approach from Nagarakatte et al. [132]. We compare it with ABOR in Figure 6-5, on the number of inserted predicate. As with more inserted predicates, runtime overhead will increase. Figure 6-5 contains a pair of bars for each test case, smaller bars are better as they represent the fewer number of inserted predicates. On average, ABOR inserted 73.87% fewer predicates than the implemented “bounds checker” method.

Figure 6-4. Comparison with a “Search & Replace” method

Figure 6-5. Comparison with a “Bounds Checker” method

6.2.2.3 Discussion

Due to the implementation limitations, for the industry systems, it is also found that ABOR caused some false positive and false negative cases. We further investigate these error cases and categorize them into three types:

Error 1- inaccuracy from alias analysis: it is difficult to implement a comprehensive alias analysis [14]. Table 6-9(a) shows a false-negative example from RounterCore that is caused by inaccurate pointer analysis. In the loop, pointer *ptr* is incremented by one in each iteration until it equals to the address of pointer *slt*. So after the loop, *ptr* and *slt* are actually aliased. However, currently we cannot detect such alias relationship. The node n2 could overwrite the value of **ptr*. Though node n1 does the boundary checking, it no longer protects node n3.

Error 2- inaccuracy from loop structure. So far only variables that are linearly updated within an iteration are handled by ABOR. Table 6-9(b) shows an example found in RounterCore. The variable *x* is non-linearly updated by using a bitwise operation. Therefore the corresponding constraint, which compares the size between *buff* and *input*, is beyond the solvability of our current implementation.

Error 3- platform-based data types: the industry programs PathFinder and RFIDScan both involve external data types of Microsoft Windows SDK (e.g., *WORD*, *DWORD*, *DWORD_PTR*, etc.). This requires extra implementations to interpret them. Additionally the value ranges of these data types specify implicit constraints. Table 6-9(c) shows a false-positive case from PathFinder. In this example, although accessing the buffer element via *index* at node n2 has no protection, there will be no buffer overflow vulnerability because *index* ranges only from 0 to 65535.

We summarize the analysis of errors in Table 6-10. In the future, further engineering effort is required to minimize the number of wrongly detected cases.

Table 6-9. Examples of errors when processing the industry prog.

(a) false-negative due to aliased pointer	(b) false-negative due to loop
<pre> void send(char mode){ // ptr and slt are two pointers while(ptr < slt){ ptr++; } /*n1*/ if(*ptr<sizeof(worduser)){ /*n2*/ cin.get(slt,80) ; /*n3*/ worduser[*ptr] =mode; } } </pre>	<pre> for(.....){ x= x & y; } char * buff = new char[x]; strcpy(buff, input); </pre>
(c) false-positive due to plat-based data type	
<pre> /*n1*/ WORD index=0; char buff[65536]; while(.....){ /*n2*/ buff[index] = cin.get(); index++; } </pre>	

Table 6-10. Accuracy and error analysis

System	Total Error	E1		E2		E3	
		#	%	#	%	#	%
RounterCore	44	23	52.27	10	22.73	11	25.00
PathFinder	26	16	61.54	0	0	10	38.47
RFIDScan	98	31	31.63	22	22.45	45	45.92
Total	168	70	41.67	32	19.05	66	39.29

6.3 Conclusion

In this chapter, we have presented an approach to remove buffer overflow vulnerabilities in C/C++ programs automatically. The main contributions in this chapter include:

- 1) We summarized the prevention rules for buffer overflow. The proposed removal mechanism enforces that the bo-sensitive sinks are accessible only after the proper validations that derived from the prevention rules.
- 2) We proposed a two-stage approach. The first stage integrates a novel buffer overflow detection method to capture buffer overflow vulnerabilities in a path sensitive manner. The second stage patches the bo-sensitive sink with the defensive code. The program is thus transformed from a vulnerable state to a version free of buffer overflow vulnerability.
- 3) The proposed method only patches vulnerable code segment. Therefore, it keeps a

lightweight runtime overhead. Using a set of benchmark programs and three industry programs written in C/C++, we experimentally show that the proposed approach is effective and scalable for removing all the detected buffer overflow vulnerabilities.

In the future, we will improve our approach and tool to minimize the number of wrongly detected cases. We will also integrate the proposed approach into some existing testing frameworks, such as CUnit, GoogleTest, to further demonstrate its practicality.

Chapter 7

RELATED WORK

Static analysis and design verification cover a broad research area with many novel approaches proposed and tools implemented. In this chapter, we group and review them from a domain oriented aspect: (1) approaches for infeasible path detection; (2) approaches for handling buffer overflow; (3) approaches for database design verification.

7.1 Detection of Infeasible Path

7.1.1 Approaches

During software developing or testing, infeasible path detection usually appears as an essential step for the final project goal. A variety of approaches have been proposed. Based on the ways that they detect infeasible paths, we classify them into six types: (1) data flow analysis; (2) path-based constraint propagation; (3) property sensitive data flow analysis; (4) syntax-based approach; (5) infeasibility estimation; (6) generalization of infeasible paths. These approaches differ in their detection accuracy, computational cost and relevantly suitable applications. We review these approaches by introducing their main features, strength and weaknesses and the related applications.

7.1.1.1 Data Flow Analysis

Classic Data flow analysis is a technique over CFG to calculate and gather a list of mappings, which maps program variables to values at required locations in CFG. Such list of mappings is called flow fact. A node with multiple in-edges is defined as a merge location, where flow facts from all of its predecessor nodes are joined together. Because of the joining operation, a variable may be mapped to a set of values instead of a single value. If a node has multiple out-edges (predicate node), each of these out-edges is defined as a control location. Flow fact is split and traversed to the successor nodes at control locations. Because of the splitting operation, a variable may be mapped to an empty set of values [104].

Data flow analysis is a common approach for detecting infeasible paths. In this type of approach, each control location would be checked when they are traversed. An infeasible path is detected when any variable is mapped to an empty set of values at a control location. In

Figure 7-1, assume we only consider the flow fact about variable *sum*. Here, *sum* is an integer variable initialized as 0. Therefore, the flow fact is initialized as [0, 0] after node1. The flow fact is traversed transparently through node2, node3 and reaching node5, after which it is split as two: one as [0, 0] flowing to node6 and the other as an empty set flowing to node7. It is then concluded that any path passing through (1, 2, 3, 5, 7) is infeasible. Approaches based on data flow analysis are often useful for finding a wide variety of infeasible paths. In the above example, the checking at node5 can detect a family of infeasible paths, which all containing the sub part (1, 2, 3, 5, 7).

Figure 7-1. Infeasible path detection with data flow analysis

However, classic data flow analysis sacrifices the detection precision, which causes some infeasible paths wrongly identified as feasible. It is important to note that the flow fact computed at a control location *L* is essentially an invariant property — a property that holds for every visit to *L*. Therefore, two things will cause the loss of the detection precision: First, the correlated branches are ignored and flow facts are propagated across infeasible paths. Second, by the joining operation at merging location, flow facts from different paths are joined, leading to further over-approximation [73]. To explain this point, consider the example program shown in Figure 7-1. The flow fact of variable *i* is initialized as [0, 0]. After node3, it is split as two: one as [0, 0] on the TRUE branch and the other as an empty set on the FALSE branch. However by simply keeping track of all possible variable values at node5, the two different flow facts are joined. The flow fact from node5 flowing to node6 is over-approximated as [0, 0]. Hence we cannot directly infer that node4 cannot be executed in consecutive iterations of the loop. Therefore, path such as (entry, 1, 2, 3, 4, 5, 6, 2, 3, 4) cannot be inferred as infeasible, which actually is. The most typical and well cited approach for

detecting infeasible paths based on data flow analysis is from Gustafsson et al. [85, 86, 88, 98]. Approaches based on data flow analysis are path insensitive.

Other similar approaches include work from Altenbernd [22], detecting infeasible paths by searching for predicates with conflict value range while traversing CFG in an up-bottom style. This approach depends on knowing execution time of each basic block in advance. The basic block refers to a piece of continuing nodes except predicate nodes. At each merge location, only the in-edge with longest execution time will be remained for further consideration. All flow facts will be checked at each control location. Those branches, which have variables mapped to an empty value set, will be detected as infeasible and excluded for further consideration. Dwyer et al. [67] proposed to adopt data flow analysis approach to check consistency violation in concurrent systems. They construct a Trace-Flow Graph (TFG), which is a modified version of CFG for concurrent systems. Variables in TFG are mapping to a set of possible system properties like sequence of event calling, synchronization of critical section. A consistency violation is found when a corresponding path is identified as infeasible in the TFG.

Approaches based on data flow analysis do not require providing a prepared set of paths. It searches for infeasible paths directly based on CFG. So they are often applied to estimate the maximum execution time for a procedure, called WCET: worst-case execution time [69] which is essential in designing real time systems. Firstly they help tighten the estimated result of WCET analysis by removing the influences from infeasible paths in the case that these paths are the longest ones. Secondly, they are useful in deriving loop upper bound in WCET analysis.

7.1.1.2 Path-based Constraint Propagation

Path-based propagation approaches apply symbolic evaluation to a path to determine its feasibility. These approaches carry a path sensitive analysis for each path in a given path set by comparing with approaches based on data flow analysis. Through symbolic evaluation, they propagate the constraint that a path must satisfy and apply theorem prover to determine the solvability of the constraint. If the constraint is unsolvable, the path is then concluded as infeasible. These approaches have high precision of detection but with heavy overhead. They are usually applied in code optimization and test case generation in which accuracy is essential. Figure 7-2 gives a general overview of these approaches.

In propagating constraint along a given path, either backward propagation [26] or forward propagation strategy [28], could be adopted to extract the path constraint under the supported data types. Forward propagation traverses the path from entry to exit and performs symbolic execution on every executable statement. Intermediate symbolic values are stored for subsequent use. It can detect infeasible paths early by detecting contradicting constraints early. It is also more straightforward and thus easier to implement, especially in the case of dealing

with arrays or containers like List and HashMap [157]. However the storage of intermediate values may grow rapidly and cause this strategy not scalable for large programs. Backward strategy applies a bottom-up traversing manner by collect path constraint first and later only search for values correlated with the path constraint. The space of the intermediate storage is largely reduced.

Based on the underlying domain theories of the constraint solver or theorem prover [10], constraint solving determines the solvability of the propagated constraint. The speed and precision of path-based constraint propagation approaches depend on the capability of the applied constraint solver. Hence, they are sound except on those cases in which existing constraint solvers have problems (e.g., floating point problems).

As mentioned by Williams [172] recently, though constraint resolution is highly efficient most of the time, it is NP-complete, and it is undecidable to know which constraint will take “too long” to be solved. Therefore, it is not always possible for these approaches to determine the feasibility of a path automatically. Furthermore, every time execution the constraint solver, there is a risk of causing the timeout exception.

Figure 7-2. An overview of path-based constraint propagation approach

Figure 7-3. An illustration of the path-based constraint propagation approach

Figure 7-3 illustrates an example of path-based constraint propagation approaches in general. Consider the target path $p = (\text{entry}, 1, 2, 3, 6, 7, \text{exit})$. After propagation along the path

using backward propagation strategy, the constraint finally becomes $((x < 2) \text{ AND } (x + 1) > 7)$. The resulting constraint is submitted to a constraint solver for evaluation. As the result is unsolvable, therefore, this path is identified as infeasible.

Among the path-based constraint propagation approaches, Bodik et al. [39] observed that many infeasible paths were caused by branch correlation and data flow analysis based approaches are overly conservative to handle them. Starting from a predicate node, they address this problem by backward propagation along a path to accumulate path constraint and consecutively evaluating the constraint with predefined rules. The path traversed is identified as infeasible if the associated constraint is determined by the predefined rules as unsolvable. If the constraint is determined by the predefined rules as solvable, the path is identified as feasible. If the solvability of the constraint cannot be determined by the predefined rules, the feasibility of the path is undetermined. Goldbeg et al. [84] addressed this problem with a more general model than others. For each targeting path, its infeasibility is determined by the corresponding path constraint. The path constraint is the conjunction of each branch condition along the path after substituting every referenced variable's definition. An independent constraint solver named KITP is invoked to evaluate the path constraint. If the path constraint is evaluated as unsolvable, then the path is identified as infeasible. Goldbeg et al. explicitly specified a domain, on which that constraint solver KITP could work. The domain limits the considered data types as Integer and Real and limits the constraint type as linear.

By equipped different constraint solvers, other approaches are able to detect infeasible paths with constraints over other domains. Ball and Rajamani [27] used a binary decision diagram (BDD) as the prover to identify infeasible paths for Boolean programs. Zhang and Wang [179] used an ad-hoc SMT solver to detect infeasible paths with path constraints over both Boolean and numerical linear domains. Bjorner et al. [38] detected infeasible paths over the domain of String by using an SMT solver called Z3 [17] which can manipulate and process those string functions in path constraints.

The advantage of the path-based constraint propagation approaches is the precision with which we can detect infeasible program paths. The difficulty in using full-fledged path-sensitive approaches to detect infeasible paths is the large number of program paths to consider. In summary, even though path-based constraint propagation approaches are more accurate for infeasible path detection, they suffer from a huge complexity.

Constraint propagation approaches are often used in areas requiring high accuracy like test case generation and code optimization. In order to generate test cases, paths will be firstly evaluated their infeasibility. Infeasible paths will be filtered. Test case generator will not process them. This helps save considerable resources and time. For example, Korel [109]

checked the path infeasibility before generating test case at the last predicate to avoid unnecessary computation. Other similar examples include the approaches from Botella et al. [41] and Prather and Myers [146]. In code optimization, def-use pairs along those infeasible paths are eliminated for enhancing the efficiency of code [39].

7.1.1.3 Property Sensitive Data Flow Analysis Approach

Both data flow analysis and constraint propagation approach have strengths and weaknesses. This section introduces the hybrid approach that combine both of them together under the framework of partial verification. The latter refers to the verification against a list of given properties to check instead of verifying all system properties. Property is an abstract term covering variables, functions or specific data structures like a pointer in C/C++.

With a given list of properties, approaches of this type have similar routine with approaches using classic data flow analysis except two modifications. First the flow fact is updated at location L only when L contains properties correlated operations. Second at merge locations, equal values for the same property from different flow facts will be merged as one; but different values for the same property from different flow facts will be separately recorded instead of joining them together. Same with approaches using classic data flow analysis, infeasible paths would be detected if any property is mapped to an empty value set at a control location. To illustrate this, let us go back to the example in Figure 7-1. Suppose variable sum and i are specified as the two properties. At the TRUE branch of node5, sum is mapped to an empty set. A family of infeasible paths containing (1, 2, 3, 5, 7) are detected as efficient as using classic data flow analysis. However, at node5, the flow facts for i will be recorded separately: fl_node3 and fl_node4 . Therefore, this time, we are able to detect infeasible paths such as (entry, 1, 2, 3, 4, 5, 6, 2, 3, 4) because only fl_node4 will be considered in the consecutive iterations of the loop.

Among this type of approaches, one well-cited work is from Das et al.[57]. They proposed an approach called ESP, whose main idea is introduced in the last paragraph, to enhance the precision of data flow analysis while also guaranteeing the verification could be controlled in polynomial time. They later extended ESP to a more abstract and general model. Work from Dor et al. [63] extended ESP for better performance over C/C++, especially for better cooperating with pointer and heap. Other similar methods may include the works from Fischer et al. [73] and Cobleigh et al. [46].

The advantage here is that this type of approaches achieves a good balance between precision and scalability. However, it brings difficulty in specifying properties accurately. There is also a risk of detection failure because of losing an accurate tracking of some properties, which having complex data structures.

7.1.1.4 Syntax-based Approach

Many infeasible paths are caused by the conflicts that can be identified from using solely syntax analysis. Syntax-based approaches take advantage from these characteristics. They capture the syntax structure of such conflicts and encapsulate them into patterns or rules. Syntax analysis is applied to detect infeasible paths through recognizing rules or patterns.

The more noticeable recent approach is proposed by Ngo and Tan [136-138]. They identified the four syntactic patterns: identical/complement-decision pattern, mutually-exclusive-decision pattern, check-then-do pattern, looping-by-flag pattern. These patterns model the obvious conflicts in a path:

- a) conflicts that arise between the value of a variable set and the value of the same variable asserted by a predicate;
- b) conflicts that arise between a set of predicates.

Let's recall Figure 2-2, the predicates at branches (2, 3) and (4, 5) are $x \geq 0$ and $x < 0$ respectively. These two predicates have an obvious conflict and can be detected from syntax analysis. Hence, the path $p = (\text{entry}, 1, 2, 3, 4, 5, 6, \text{exit})$ in Figure 2-2 is clearly infeasible. Based on the four patterns identified, they developed an approach to detect infeasible paths from any given set of paths. Through the use of these patterns, the approach can avoid the costly symbolic evaluation by just using solely syntax analysis to identify infeasible paths through recognizing these patterns. In opposing to other approaches, their approaches were proposed as an independent approach. They have also conducted an experiment on code from open-source systems and found their approach can detect a large proportion of infeasible paths.

Among those earlier syntax-based approaches, one well-cited work is from Hedley and Hennell [92]. They proposed to use heuristics rules to detect four types of infeasible paths: infeasible paths caused by consecutive conditions, infeasible paths caused by inconsistency between test and definition, infeasible paths caused by constant loop control variable and infeasible paths caused by constant loop times. These rules are quite efficient as they solely based on syntax analysis. Later Vergilio et al. [164] showed that, by a fast scan, Hedley and Hennell's rules were able to identify nearly half paths as infeasible in a path set with 880 sample paths.

The advantage of syntax-based approaches is that they can avoid the expensive symbolic evaluation by applying solely syntax analysis to achieve some efficiency. However, these approaches may report a small number of false positives as they just based on syntax analysis alone, that is, they suffer from the possibility of reporting false-positive results.

Syntax-based approaches detect infeasible paths from a set of paths, so they rely on well-constructed paths set. They are often used for a fast scan during the early testing stage. They are also used as the first step for code optimization or coverage estimation, to avoid the influence of infeasible paths during the later analysis.

7.1.1.5 Infeasibility Estimation Approach

Early researchers have found the problems caused by infeasible paths, and it was very hard to achieve a satisfied detecting result. Therefore, they managed to build statistical metrics to estimate the number of infeasible paths in a given procedure based on certain static code attributes. The most famous work is from Malevris et al. [125]. They stated “the number of predicates involved in a path being a good heuristic for assessing a path’s feasibility”. The greater the number of predicates exist in a path, the greater the probability for the path being infeasible. They further concluded a regression function $f_q = K^e - \lambda_q$ to represent the above relationship, in which K and λ are two constants, q stands for the number of predicate nodes involved in a given path while f_q stands for the possibility of this path being feasible. Vergilio et al. [164] validated the above results over a broader selection of programs and extended the work to involve in more static code attributes, such as number of nodes, number of variables and number of definitions.

The advantage of such metrics is that they are easy to implement and provide a fast way to predict path infeasibility within a confidence level [164]. However, it is an approach of rough estimation rather than accurate detection. The accuracy of the regression function also biased over different test programs and different programming language.

7.1.1.6 Generalization of Infeasible Paths Approach

When a path is infeasible in a CFG, all other paths that contain the path are also clearly infeasible. Based on this simple concept, Delahaye proposed to generate all infeasible paths from a given set of seed infeasible paths [58]. It provides a convenient way to generate error seeded models for further testing, especially for legacy programs or combined as a component in regression testing.

7.1.2 Tools Implementation

In most program optimization, software analysis and testing tools, infeasible path detection usually appears as an important component. Best to our knowledge, there is no independent tool particularly designed for it. In this section, we introduce related existing tools based on above approaches. We also brief the implementation details of the reviewed approaches to help those who want to detect infeasible paths in their own applications.

Table 7-1. Useful Tools for Implementation

Name	Description	Link	Supported Language
Code analysis and optimization			
Soot	A Java Optimization Framework.	http://www.sable.mcgill.ca/Soot/	Java
CodeSurfer	Code analyser	www.grammatech.com/products/CodeSurfer/	C/C++
Pixy	Code and security analyser for PHP	http://pixybox.seclab.tuwien.ac.at/pixy/	PHP
Automated Test Case Generation			
CUTE	Test case generation for C/C++	http://cute-test.com/wiki/cute/	C/C++
jCute	Test case generation for JAVA	http://cute-test.com/wiki/jcute/	JAVA
CREST	Test case generation for C	http://code.google.com/p/crest/	C
Pex	Structural testing framework for .Net	http://research.microsoft.com/en-us/projects/pex/	C#, C++, VB.net
eToc	Path selection with genetic algorithm and test case generation for C/C++, Java	http://www.informatik.hu-berlin.de/forschung/gebiete/sam/Forschung%20und%20Projekte/aktuelle-forschung-und-projekte/softwarekomponenten-entwicklung/eodl-projects/etoc/etoc	Java, C/C++
Theorem Prover			
Z3	SMT prover	http://research.microsoft.com/en-us/um/redmond/projects/Z3/	C/C++
Lp_Solver	Linear constraint solver	http://lpsolve.sourceforge.net/5.5/	C/C++, JAVA, PHP, Matlab
BLAST	Lazy abstraction software verification	http://mtc.epfl.ch/software-tools/blast/index-epfl.php	C
Verification and Error detection			
ESC-Java	Error checking for annotated Java program	http://secure.ucd.ie/products/opensource/ESCJava2/	Java
SLAM	Verify critical properties for C/C++	http://research.microsoft.com/en-us/projects/slam/	C/C++
LCLint	Static analysis for C/C++	http://www.splint.org/guide/sec1.html	C/C++

We select 14 relevant tools and analyse them in this section. These tools are summarized in Table 7-1.

7.1.2.1 Data Flow Analysis Approach

The most completed work is from Gustafsson et al. [85, 86, 88, 98]. In order to detect infeasible paths over complex programs, they decompose a program into several linked scopes. The latter is a set of program statements within a call of one procedure or an execution of a loop iteration. It is statically created so that calls to a function or a loop at different call sites will be marked and analysed separately. This brings in high precision but requires expensive computation cost. The scope graph is a hierarchical representation of the structure of a program which is used to describe the interaction relationships between scopes. Data flow analysis based on abstract interpretation will be performed over each scope separately to compute the live variables set. A recorder is created to store the infeasible paths detected within each scope. Among the work of Gustafsson et al., only primary data types are mentioned. There has been no corresponding open source toolkit published. For readers planning to code based on this type of approach, they could utilize existing data flow analysis framework to find out variables mapped to an empty value set at certain control locations and detect infeasible paths accordingly. For example, the sub package Spark in Soot can perform intra or inter data-flow analysis over Java procedures. Other available toolkits are, for example, CodeSurfer for C/C++, Pixy for PHP.

7.1.2.2 Path-based Constraint Propagation Approach

Usually these approaches appear in tools of automated test case generation. The typical example is concolic testing [151]. Before test data is generated, the target path will be tested its infeasibility by submitting the path constraint to theorem prover. If it is infeasible, the last predicate condition will be reversed to stand for a new path containing the opposite branch. The details could be found in the following tools: CUTE and jCUTE which are available as binaries under a research-use only license by Urbana-Champaign for C and JAVA; CREST, which is an open-source solution for C comparable to CUTE; Microsoft Pex, which is publicly available as a Microsoft Visual Studio 2010 Power Tool for the NET Framework.

Readers, who are interested in implementing this type of approaches in their own application, can first apply those data flow analysis tools to propagate along a path and generate path constraint in symbolic expressions. Later the constraint could be submitted to theorem provers. Available tools of the latter include Z3, an SMT solver from Microsoft; Lp_Solver, an open source linear constraint solver under GNU license; BLAST, which is a checker often used to verify abstract program.

7.1.2.3 Property Sensitive Data Flow Analysis Approach

Das et al. [57] proposed a tool called ESP, which is a research project for partial verification under Microsoft. ESP is able to construct CFG and perform property sensitive analysis in either intra-procedure or inter-procedure mode. The verification for given properties at control location is handled by its build-in rules for primary data types in C/C++. However, the tool provides an interface to replace the build-in rules with standard theorem provers for more complex analysis. There are also several other tools based on this type of approaches, such as ESC-Java, SLAM, LCLint.

7.1.2.4 Syntax-based Approach

Syntax-based approach is easy to implement because the heuristics are concluded from code and expressed in a straightforward style for implementation. Ngo and Tan [136] implemented an automated test case generation system called jTGEN for automated test data generation for Java. The system consists of an infeasible path detector that based on heuristic code-patterns, a code parser based on Soot, a path selector and a test case generator based on eToc. The system uses a genetic algorithm to select a set of paths. These paths are checked against heuristic code patterns and only feasible paths will be remained for test case generation.

7.2 Buffer Overflow Handling

This section briefs the three classes of methods for buffer overflow detections: (1) the first type is to create runtime defence to prevent exploiting potential vulnerabilities of any installed programs; (2) the second is to analyse codes and find out the vulnerabilities in a program; (3) the last is to try to removal the vulnerability from the original code.

7.2.1 Method of Static Analysis

Buffer overflow vulnerability is detectable by finding variables, which are exploitable by users to violate certain buffer's boundaries. In order to manage the balance between soundness and scalability, static analysis based detection could be further classified into path sensitive approaches and non-sensitive approaches.

Path sensitive approaches propagate and analyse a given path to generate path constraint according to the properties which ensure the path are not exploitable for any buffer overflow attack. The path constraint is extracted using symbolic evaluation through either forward or backward propagation. Constraint solver or theorem prover is applied to evaluate the constraint. If the constraint is determined as unsolvable, then the path is concluded as vulnerable. These methods pursue the soundness and precision but include heavy overhead due to the use of symbolic evaluation.

Xie et al. [174] created ARCHER, a static checker to capture memory access violation in C programs. The tool parses source code into AST tree; generates corresponding CFG and extracts paths by traversing the CFG using a straightforward depth-first search. ARCHER implements two heuristics to handle loops. The first strategy is only to compute value ranges for those variables that are used in the loop predicate. The second strategy is to go through the loop for only once. Symbolic evaluation is performed on those paths containing suspected memory access and generates path constraint by considering assignment, predicate conditions, and effects of function calls. The path constraint is sent to an ad-hoc constraint solver to judge whether buffer boundaries would be exceeded under this path constraint. A similar path-sensitive approach is proposed by Cova et al. [51] to find vulnerabilities in x86 executable binary files. They also apply taint analysis to focus on sink statements related with user input.

Le and Soffa proposed Marple [113] [114], a path-sensitive analyser, to give a more comprehensive detection result. The method could identify a path as either of *infeasible/safe/vulnerable/overflow-input-independent/don't know*, by raising queries and propagating it backwards along the control flow and updating it at nodes where information can be collected. The evaluation ends when the query is solved, or the entry node is reached. This state-of-the-art approach in vulnerability detection area has the following features:

- 1) It is reliable and stable
 - It detects vulnerabilities (e.g: buffer overflows) in a path sensitive manner, which offers high precision.
 - It is extended on a classic static analysis framework from Bodik et al. [40], which provides mature solution for inter-procedure analysis.
- 2) It is scalable
 - It is demand-driven, which means it only analyse those suspicious paths.
 - It handles encountered loops by abstracting their impact to variables which referenced by the sink and updated linearly by the loops.
 - It follows Microsoft Phoenix infrastructure to handle alias in an intra-procedural-based, field-sensitive and flow-sensitive way.

However, the analysis during the propagation and query resolving may lead to heavy cost. And due to runtime complexity, Marple may still generate many false positives.

Non-sensitive approaches try to avoid complex analysis. They normally rely on general data flow analysis. These methods compute the environment, which is a mapping of program variables to values, at those suspected locations. The environment captures all the possible values at a location L . Therefore, if there is any value violating the buffer boundary at L ; it

stands for a vulnerability of buffer overflow. To choose proper location and make safe approximation requires heuristics and sacrifices precision. However, such design wins more scalable performance, which is essential when dealing with large-scale applications.

Several approaches use annotations to narrow the search space. Users need to mark annotations like pre- and post-conditions of a statement. Data flow analysis is carried out to calculate the point-to information only at the annotated statement. CSSV [64] is such a typical system. It transforms a C program into an integer program, and static analysis is performed to detect faulty integer manipulations. Buffer overflow vulnerability is detected by finding an invalid pre-condition before an annotated statement. However in order to generate sound approximation of pre-conditions, CSSV still involved in heavyweight and performed poorly to large programs. Larochelle and Evans [112] created another annotation-based method with higher scalability by sacrificing more precisions including: only consider the system library functions; only consider the first and last iteration of a loop. The main shortcoming is that these methods require users' rich experience and considerable manual effort.

Instead of using annotations, Wagner et al [167] heuristically choose a series of C statements that can be exploited by attackers. Each of such statements is interpreted as an integer constraint, which is the safety property ensuring this statement is free from the buffer overflows. A general data flow analysis is performed to substitute relative variables' value into those integer constraints. An ad-hoc integer solver then evaluates these constraints. The similar work is from Ganapathy et al [82]. They only focus on the buffer overflows caused by C string functions. Each function is transformed into a linear integer constraint. A flow-insensitive taint analysis is performed to eliminate constraints that are not related with input variables. Therefore, the scalability is further improved.

Hackett et al [89] defined more abstract models. Similarly, a set of statements are chosen and assigned with constraints that could ensure the statement free from buffer overflow attacks. The constraints are inserted back to the code to generate a new program. Therefore, it is transformed to prove the reachability of those locations after those inserted constraints. The reachability can be proved by a number of popular model checkers. Wang et al [115] proposed similar method but improved the scalability by first slicing the program, leaving the statements related with those constraints.

Saxena et al [150] proposed a loop extended symbolic evaluation framework to perform flow-sensitive analysis over paths with loops. This framework focuses to address the scalability issue for analysing loops. They abstracted a loop as a separate unit and connect the inner and outside environments by two steps: (1) Variables modified within this loop are linked with trip count, an abstract value representing the loop times; (2) trip count is further

linked with input variables. Their design provides a more accurate and scalable way to find buffer overflows on paths with loops. However, this method currently could only handle variables that are linearly updated within each loop iteration.

7.2.2 Method of Runtime Defence

These approaches aim to provide an additional protection regardless of what the source program is. Such protections are usually implemented from three different kinds of aspects: code instrumentation, infrastructure modification and network assistance.

The first one is to partially simulate a runtime environment, execute the suspected program and check whether malicious actions have taken place: Stake Guard [52] creates a canary before return address; RAD [171] creates a global array as a copy of functions' return addresses.

The second case is to modify the under layer mechanism to eliminate the root of attacks: Xu et al. [175] proposed to split stack and store data and control information separately; Ozdoganoglu et al. [142] proposed to implement a non-executable stack. The last one is to maintain a black list recording the sources which often send suspicious data. More details and corresponding tools of runtime defence could be found in a recent survey [36].

7.2.3 Method of Attack Prevention through Auto Patching

Few of the current techniques focus on removing the buffer overflow vulnerabilities by adding defensive code automatically. In principle, the defensive code could ensure that a program is free from desired attacks due to vulnerabilities. This kind of approaches validate sink node to ensure no taint data could impact the sinks. They have shown the practicality in areas such as the removal of cross-site scripting vulnerabilities [37, 100, 101, 152]. The only similar work on buffer overflow vulnerability removal is from Lin et al. [117]. Though their approach also repairs some buffer overflow vulnerability on the ad-hoc basis, they inserted a general check for any sink node based on path-insensitive information. Hence in opposing to the proposed method, the check inserted may not preserve the original program behaviour or may not be sufficient to all cases.

7.3 Database Design Verification

Currently, majority of the methods for design verification are either (1) formal method or (2) general concept based approaches.

The former methods require the specifications to be defined formally [48, 49, 53, 74, 93, 94, 165] [139]. The effort and time required to produce formal specifications makes these methods

more suitable for only safety critical systems and not suitable for general database applications. Moreover, the former methods usually rely on UML tools and require strong expertise. This prevents its wide adoption in early stage of database development.

The second kind of approaches use general concepts such as correctness, completeness and consistency to verify a specification. They do not offer further details on specific designs. Furthermore, all these approaches do not offer details from a domain-specific perspective as they are general approaches. Though the remaining design verification approaches could be domain-specific, but they also do not address the design for the interaction between a database application and its database [23, 126]. These approaches have a similar idea with data flow anomalies detection [96], which identifies certain general program errors, such as dead code, uninitialized variables, write-write confliction. But there are several fundamental differences between them:

- 1) The general concept based database design verification is working on textual specification and database schema while the data flow anomalies detection is working on the source or binary codes of programs.
- 2) The general concept based database design verification relies on CASE tools and text mining tools. The data flow anomalies detection relies on dataflow analysis tools or compilers.
- 3) The domain knowledge required is largely differently.

7.4 Comparison and Summary

In this chapter, we have reviewed a series of novel techniques in four domains: (a) infeasible path detection; (b) buffer overflow detection; (c) buffer overflow removal; (4) database design verification.

However existing approaches are not able to resolve these domain issues perfectly. To chase analysis precision, current approaches mainly apply full symbolic evaluation instrumented with constraint solvers. Symbolic evaluation is inherently computational expensive. Furthermore, theoretically, constraints imposed from static analysis can be in any form. It is believed that program complexity will highly increase the difficulty of solving these constraints. It needs considerable resources to develop a general model to cover them.

As the above four domain issues could be viewed as a model abstraction and verification. Snifakis suggested [44, 153] that general verification theory would be of theoretical interest only. By contrast, a compositional study of particular classes of properties or systems would be highly attractable, which indicates to introduce in empirical knowledge. Clearly, systematic experiment instrumented with both automated and manual evaluation to discover and validate

empirical knowledge is very difficult and requires high cost of time. However, researchers should still consider spending effort on this.

In this thesis, we follow this guidance to investigate the characteristics of constraints imposed from control flow graph or use cases. With extensive experiment and manual work, we empirically identify attractive characteristics that the majority of these constraints are possessed. We took advantage from these characteristics to establish good practical approaches to improve the state of the art on both the accuracy and scalability. More specifically, we have done the work as following.

In the area of database design verification, the existing works mostly focus how to use formal methods to describe entire software from a general perspective. Their works [48, 49, 53, 74, 93, 94, 165] [139] tend to be formal and general, which requires additional expertise to interpret and adopt them. This limits their usage in practice. On the contrary, we focus on discovering domain knowledge of database design. The discovered domain knowledge is combined with formal methods to achieve better performance. This combination generates a new approach which can automatically capture most design faults in the early database design stage.

In the area of detecting infeasible paths, we have proposed DIPLE, a pattern-based approach to detect infeasible paths. DIPLE is also based on constraint propagation. But as opposed to other constraint propagation approaches, based on the two infeasible path patterns, DIPLE uses only syntax analysis without any symbolic evaluation to filter away paths that will never match any of the two patterns. For the remaining paths, DIPLE applies only limited symbolic evaluation instead of full scope symbolic evaluation. We have implemented DIPLE using Soot and Z3. In our extensive experiments, putting aside the floating-point rounding problem in Z3, DIPLE shows high accuracy. Our experiments also showed DIPLE filtered a large proportion of paths using only syntax analysis without any symbolic evaluation. All these results, therefore, suggest that DIPLE is much more accurate, complete and efficient than other approaches based on constraint propagation.

More specifically, DIPLE generalizes and extends the rules for detecting infeasible paths employed in existing techniques including but not only those proposed in [39, 151, 162, 173]. DIPLE does so by generalizing and extending existing methods to characterize most infeasible paths in real system code as two general patterns. With such characterization, DIPLE is able to detect a large scope of infeasible paths accurately and efficiently.

Clearly, DIPLE can also be implemented using syntax analysis and a model checker [45] instrumented with any theorem prover. Hence, our method is not meant to compete with constraint solvers, theorem provers or model checkers. Indeed, based on the recognition of the

two infeasible path patterns, our method also provides a novel way to improve the use of these techniques for the detection of infeasible paths. It can also be viewed as a better way to use these techniques through experimental algorithms [71, 127].

DIPLE has evolved from our earlier method proposed in [136]. It relied on heuristics to detect paths that are probably infeasible (approximate method). This yielded significant numbers of false-positives. DIPLE has made enhancement on correctness, completeness and efficiency.

In the area of handling buffer overflows, we proposed two approaches. One is a pattern based approach to detect buffer overflows. Our proposed method focuses on scalability while not sacrificing the precision of detection by making limited use of the theorem prover Z3 to selected paths and branch predicates. This selection is made according to the two patterns that characterize most of the paths, which contains buffer overflow vulnerability, in real system code. Hence, it is also a form of path sensitive analysis. As opposed to other works, our work contributes to improving (1) Model the common buffer overflow vulnerability into three codes patterns. Pattern identification is done through syntax analysis and a light-weighted restricted symbolic evaluation. (2) Present two relationships between branches, helping to further enhance the scalability. (3) The experiment showed the high detection precision and scalability of the proposed pattern and detection method.

Another approach we proposed is ABOR, a pattern based approach to removal buffer overflows automatically. Our approach follows an existing research direction, which is to locate the vulnerability first and then transform the vulnerable code segment into a safe version. This helps reduce the size of added code without compromising the removal precision. Moreover, ABOR detects the buffer overflows based on path sensitive information and then only add defensive code to repair the vulnerable code segment. Therefore, it adds limited code and has a low runtime overhead.

Chapter 8

CONCLUSION AND RECOMMENDATIONS

This chapter summarizes the contributions of this research and discusses the limitations of our current work. It also suggests future work that would improve current work and integrate the proposed approaches with existing software engineering activities.

8.1 Conclusion

Static program analysis and database design verification are crucial since they could help improve software quality and development efficiency. Static program analysis and design verification are key areas in software engineering. Though many approaches and tools have been developed, they are not widely adopted in current software industry. The inadequacy of existing techniques and is that: it is difficult to find suitable abstraction models which could manage both scalability and accuracy when facing large systems.

Empirical knowledge is essential as it is learnt from experts' practices and extensive experiments. Applying empirical domain knowledge to software engineering is a bright direction. Successful cases include design patterns, code metrics, etc. In this thesis, we followed guideline of empirical study to discover a set of empirical patterns in the areas of Static program analysis and design verification. With these patterns, we augmented static program analysis and design verification with empirical domain knowledge. The knowledge is encapsulated as verification patterns. We proposed four approaches to address the following problems with pattern recognition.

Database design verification: Database design is essential but existing approaches for database design verification are either too general or too formal. Therefore in practice, it is more relying on senior programmers' experience and manual effort to verify the design. However, such experience and effort are usually kept and shared in an informal way, such as blogs on websites.

Our objective is to study such experience and encapsulate them into empirical verification patterns. Based on our study so far, we found data maintain in a database are usually dynamic and changeable. In most cases, both attribute and record type have a lifecycle that starts from being inserted, via being used and optionally being updated in any number of times and sequences, to being deleted.

We proposed a domain-specific design verification approach for the development of database applications. The approach is based on a factor that in any database applications, data may have a lifecycle. We extracted patterns from such factor and propose to use these patterns to identify errors in database design. Through two cases studies, we find it useful in detecting key design errors such as missing essential functionalities. Our approach targets the early stage verification as at that time the error can be fixed with much lower the cost.

Infeasible path detection: Infeasible path detection plays a vital role in software testing, maintenance and optimization. After reviewing the existing approaches, we found none of them can detect a majority of the infeasible paths efficiently. The goal was to improve the current state of the art by making the detection of infeasible paths in real systems more accurate, complete, and efficient. We proposed two empirical patterns over CFGs that lead to infeasible paths: flag-based inhibitor and conflicting branch patterns.

We also proposed the DIPLE to detect infeasible paths based on the two patterns. In order to be scalable to large systems, DIPLE recognizes patterns through syntax analysis and a light-weighted limited symbolic evaluation. DIPLE is designed to detect infeasible paths from a given set paths on both intra-procedure and inter-procedure level.

A prototype system has also been implemented to assess DIPLE empirically through an extensive experiment. Experiment shows that, regarding Java code of selected real-systems, all the infeasible paths detected by DIPLE were actually infeasible (correctness) and DIPLE detected all the infeasible paths (completeness) that we could identify with the help of Z3 when possible and manually otherwise. The experiments also show the scalability of DIPLE when applying to large systems.

Buffer overflow detection: buffer overflow is common. It exists in both legacy system and current products from leading software companies (Google, Microsoft, etc.). Buffer overflow is among the top dangerous vulnerability. Attacker can easily hijack the system after exploiting one of such vulnerability.

Our goal was to improve the current state of the art by making the detection of buffer overflows in real systems more efficient and scalable. We identified and proposed three patterns that cause buffer overflow vulnerabilities: pattern of vulnerable syntax usage, pattern of vulnerable element accessing and pattern of vulnerable bulk moving. A detection method is proposed to identify vulnerable paths through recognizing the three patterns. The pattern recognition is using syntax analysis and a restricted symbolic evaluation to enhance the scalability.

We carried experiments to compare our approach with another popular method to evaluate its accuracy and scalability. The results show the proposed approach detected all of those

reported buffer over vulnerabilities. It also showed that, on average, the proposed approach saved 26.47% of the processing time, visited 37.82% less of the processed CFG nodes and 17.96% less of the processed predicate nodes, invoked theorem prover 73.68% times fewer, and traversed 58.60% fewer paths.

Buffer overflow removal: current efforts that address buffer overflow problem mainly focus on providing effective detection method or creating a protective run-time environment. However, as sophisticated attack vectors evolve, it is also necessary to have methods to clean up those detected vulnerabilities from the original program automatically.

Our goal is to provide a solution as supplementary to existing techniques, which is efficient and scalable to automatically remove buffer overflow vulnerabilities from C/C++ applications. We reviewed the-state-of-art in buffer overflow detection and proposed four empirical rules which could prevent the occurrence of buffer overflows. A framework, ABOR is proposed to eliminate buffer overflows once detected. The proposed removal mechanism enforces that the bo-sensitive sinks are protected by proper validations that derived from the prevention rules. The approach consists of two stages. The first stage extends a novel buffer overflow detection method to capture buffer overflow vulnerabilities in a path sensitive manner. The second stage patches the bo-sensitive sink with the defensive code. The program is thus transformed from a vulnerable state to a version free of buffer overflow vulnerability.

We have implemented ABOR and evaluated its effectiveness through experiments on seven benchmarks systems and three industrial C/C++ applications. The results show that the proposed approach worked effectively to remove all the detected buffer overflow vulnerabilities in the studied subjects.

8.2 Limitation of the study

In this thesis, a number of empirical verification patterns and related recognition approaches are proposed. The accuracy and scalability of proposed approaches are validated through extensive experiments. Though our study involves many systems from different sources and much effort has been directed to address the correctness and generality of the experiment results, the following issues may still pose threats to validity and requires future efforts when resources are available.

- 1) The accuracy of our evaluation results is subjected to the accuracy of instrumented theorem prover. In this thesis, Z3 from Microsoft is applied. Though Z3 is a solver with the latest techniques in solving Satisfiability Modulo Theories (SMT). It has limitations in floating-point computations, which may cause potential rounding errors. It also may fail to solve complex nonlinear constraints. We involved manual work to minimize the possible

errors caused by this.

- 2) In order to evaluate the practicability of the proposed approaches thoroughly, we should test them on systems written in different programming language, which requires us to develop a different prototype analysis tool. But due to the substantial effort involved in such development, all our experiments were performed on Java code.
- 3) Though the scale and scope of our study are quite large, there is still room for further improvements regarding generality, including increasing the number of randomly selected paths and industrial systems, and covering more application domains and levels of system maturity (pre-unit-testing stage, operational stage, etc.).
- 4) The impact of programmers' skills and practices in implementation may also require further examination.
- 5) In the experiment, manual work is involved for further validation. Though our manual work was carried out by our team members independently and reconciled when inconsistent, no complex manual process is fully error-free.

8.3 Future work and recommendations

We believe that our way in addressing the domain problems— characterizing the target objects in real systems to overcome theoretical limit (e.g.: constraints are in general unsolvable) such that a practical solution can be found in the presence of the limit — could be an important direction to explore for addressing other issues in software analysis and testing such as security vulnerabilities and faults detection. Though there are still limitations with the proposed approaches, we would like to put efforts in finding practical solutions to improve our current results and enhance our work presented in this thesis. More specifically, it includes:

- 1) **Empirical study of constraints in code:** Constraint solving is a fundamental technique in program analysis to solve the constraint imposed by the branches in a path. However, in general, solving constraints faces the well-known intractable problems, such as floating-point issues, integer overflow problems, and nonlinear equations. We are currently studying the hypothesis characteristics of constraints in program code using data mining techniques. Our preliminary study reveals that much of the constraints need to be solved in program analysis are linear constraints. It also reveals that with proper substitution of function calls with variables, indeed, much of the constraints can be solved by existing constraint solving techniques. An early stage work of this has been integrated into a crash reporting project, which has been submitted to ASE'2014 [187].
- 2) **Language independent component:** domain problems vary with each other. They might be a common issue for several program languages or also possible as a unique

vulnerability over one popular language. On one hand, buffer overflow vulnerability is unique for programs written in C/C++, but on the other hand infeasible paths detection and database design verification are independent on programming languages. For the latter cases, all the proposed approaches in this thesis are implemented with Java and Soot. Soot provides Java-based CFG representation. We need to equip the prototype systems with a component that could translate source codes in different languages into one language-independent CFG representation. All the proposed algorithms should be implemented over the new CFG representation.

- 3) **Overcome the path explosion issue:** Software grows rapidly in both size and complexity in the current trend, which may produce a large number of paths and constraints for analysis. For approaches of path sensitive analysis, which include our proposed approach for infeasible path detection, buffer overflow detection and removal, there is a danger to meet tremendous amount of paths that overpower the available computational resources. To improve our approaches to solve this, another possible direction is to apply intelligent approach like genetic algorithm to guide the path selection [159]: by choosing proper fitness function, only paths with high suspicion of infeasibility would be kept for further processing. Another idea is from Forgács and Bertolino [75] who utilized program slicing. They reduced a program to a slice of the variables and statement concerned and thus simplify the detection of infeasible paths. Last but not the least, it would be a good attempt to use multi threads to process the paths concurrently.
- 4) **Integrate with existing tools to support external dependency:** in an empirical study, there requires extensive manual work in identifying and constraining those nondeterministic behaviours. These nondeterministic behaviours are possibly caused by the following factors:
 - a) There are function invocations when the function code is not available.
 - b) There are additional constraints of certain system function calls, such `six(x)` is always within `[-1, 1]`.
 - c) There is information from local resources (e.g.: property files) while the analysis tool is not granted the access right.

To overcome these, currently we are preparing to integrate the proposed patterns and approaches with unit testing frameworks include JUnit and CUnit. This would bring in two main benefits. First, the analysis can utilize the existing mock objects library of these frameworks. This helps greatly reduce the incidence and negative consequences from those objects with nondeterministic behaviours. Second this effort also makes the proposed approaches towards a more practical and applicable implementation for industrial.

- 5) **Integrate with existing tools to improve their performance:** we would like to integrate our research work in the thesis into existing tools to improve the state-of-the-art. Several initial work has been done:
- a) We study more vulnerability that are similar with buffer overflows and categorize them as a type of “input manipulation” vulnerabilities. We integrate the detection techniques with data mining techniques for better vulnerability detection accuracy. The initial work has been published in APSEC’2013 [188] and an extended work is under review of IET Software [185].
 - b) We study how to use the result of infeasible path detection to refine code coverage analysis. The initial work has been published in CSMR-WCRE’2014 [183].

APPENDIX A – RESULTS OF EXPERIMENT IN

CHAPTER 4

The tables and figures in Appendix A1, A2 and A3 report the results for our BP0, CtP2 and Random experiments, respectively.

The identification of each table follows the format X.n.m, where X denotes the experiment name, n denotes the type of paths experimented, and m denotes the type of experiment results. For experiment name, we use “A1”, “A2” and “A3” to represent BP0, CtP2 and Random experiments, respectively. For type of paths experimented, we use “1” and “2” to represent experiments on intra-paths and inter-path, respectively. We organize our experiment results into four types: “1” represents size statistics about the various systems and their CFGs; “2” represents the results obtained when using exhaustive symbolic evaluation; “3” represents the results when using DIPLE; “4” represents the results for comparing DIPLE and exhaustive symbolic evaluation on false-positives, false-negatives and processing time, including statistical testing for processing time; “5” represents the result for the classification of additional infeasible paths detected by DIPLE. For example, Table A1.2.3 shows the results of the BP0 experiment on inter-paths using DIPLE.

In all the tables, “#Path” denotes the total number of paths in the context specified. In the tables for experiment results of type 1, columns “#Proc” and “#Nodes/path” show the total number of procedures and nodes per path in a system, respectively. “Stdev” stands for standard deviation. In the tables for experiment results of type 2 and 3, “#infP” shows the total number of infeasible paths in the context specified. In addition, in the tables for experiment results of type 2, the sub-column “#notInfP” under the shows the total number of paths that were automatically evaluated as not infeasible. In the tables for experiment results of type 3, in the column “#infP Detected”, sub-columns “Pattern 1”, “Pattern 2” and “Sub-infP” show the number of infeasible paths detected by Pattern 1, Pattern 2, or by the fact that it contains an infeasible sub-path detected earlier, respectively. The column “Paths processed by Syntax Analysis only” shows the number (#) and percentage (%) of paths that are filtered away by DIPLE solely using syntax analysis without any symbolic evaluation. The column “% Pred Nodes Eval” shows the average percentage of predicate nodes per path included in the *l*-constant propagation or *l*-symbolic evaluation in DIPLE. To address the generality of our evaluation results across application domains, we selected a sizable number of open-source systems from a variety of application domains [5, 8, 11, 18].

A1 – Results of BP0 Experiment

Appendix A1 shows the tables and figures for the results of the BP0 experiment.

Table A1.1.1. Size statistics of the BP0 experiment on intra-paths

System	Size (KLOC)	#Proc	#Path	#Nodes/path	
				Mean	Stddev
Open Source Systems					
JCM/Math Lib	12.28	908	3368	46.74	94.97
InfinityPFM	6.91	675	1290	24.21	40.10
Struts/WAF	18.46	825	3370	13.72	23.11
ILedger	8.79	1816	2423	10.80	15.78
JasperReports/RE	106.63	10709	19603	23.62	43.42
Regatta	9.71	1685	2237	11.76	23.94
JSpider	5.34	1406	1295	13.26	14.80
JTidy	13.37	768	3108	69.69	93.61
Total (Open Source)	181.49	18792	36694	25.37	47.08
Industry Systems					
Blog	12.44	1602	8558	20.43	34.81
OfficeAssistant	8.23	1091	1195	17.55	15.59
ObjectLibrary	7.34	531	848	14.14	30.90
Redis	4.13	337	1041	17.85	19.45
Total (Industry)	32.14	3561	11642	17.85	32.38
Error Seeded Systems					
JMeter/Load Test	17.74	2239	4016	18.55	27.09
XML-Security	14.74	1310	3031	17.07	18.36
Total (Error Seeded)	32.48	3549	7047	17.91	24.95
Grand Total	379.72	37687	83506	22.74	43.54

Table A1.1.2. Results of the BP0 experiment using exhaustive symbolic evaluation on intra-paths

System	#Path	exhaustive symbolic evaluation		
		#Path	#infP	#notInfP
Open Source Systems				
JCM/Math Lib	3368	2155	924	1231
InfinityPFM	1290	916	165	751
Struts/WAF	3370	2839	100	2739
ILedger	2423	1848	126	1722
JasperReports/RE	19603	12163	1667	10496
Regatta	2237	1786	53	1733
JSpider	1295	897	63	834
JTidy	3108	2381	1203	1178
Total(Open Source)	36694	24985	4301	20684
Industry Systems				
Blog	8558	5134	616	4518
OfficeAssistant	1195	760	132	628
ObjectLibrary	848	719	35	684
Redis	1041	609	108	501
Total(Industry)	11642	7222	891	6331
Error Seeded Systems				
JMeter/Load Test	4016	2475	389	2086
XML-Security	3031	1737	366	1371
Total(Error Seeded)	7047	4212	755	3457
Grand Total	55383	36419	5947	30472

Table A1.1.3. Results of the BP0 experiment using DIPLE on intra-paths

System	#Path	#infP Detected				Paths processed by Syntax Analysis only		% Pred Nodes Eval
		Pattern1	Pattern2	Sub-infP	Total	#	%	
Open source Systems								
JCM/Math Lib	3368	195	217	512	924	2237	66.42	25.55
InfinityPFM	1290	45	61	59	165	993	76.98	36.48
Struts/WAF	3370	24	39	37	100	3207	95.16	18.36
ILedger	2423	42	42	42	126	2212	91.29	24.56
JasperReports/RE	19603	488	284	895	1667	17741	90.50	14.16
Regatta	2237	10	20	23	53	2117	94.64	21.99
JSpider	1295	27	16	20	63	1158	89.42	29.55
JTidy	3108	100	621	482	1203	1441	46.36	43.47
Total (Open Source)	36694	931	1300	2070	4301	3110	84.77	23.91
Industry Systems								
Blog	8558	239	149	228	616	7567	88.42	25.71
OfficeAssistant	1195	39	47	46	132	941	78.74	45.00
ObjectLibrary	848	19	2	14	35	813	95.87	19.90
Redist	1041	35	36	37	108	841	80.79	31.36
Total (Industry)	11642	332	234	325	891	10162	87.29	27.77
Error Seeded Systems								
JMeter/Load Test	4016	161	78	150	389	3449	85.88	18.73
XML-Security	3031	175	70	121	366	2286	75.42	40.68
Total (Error Seeded)	7047	336	148	271	755	5735	81.38	28.17
Grand Total	55383	1599	1682	2666	5947	47003	84.85	24.80

Table A1.1.4. Results for comparing DIPLE with exhaustive symbolic evaluation from BP0 experiment on intra-paths

System	InfP Detected							Avg Proc Time Per Path (ms)	
	DIPLE			exhaustive symbolic evaluation					
	#Correct	#False - Positive	#False - Negative	#Correct	#False -Positive	#False-Negative		DIPLE	exhaustive symbolic evaluation
#						%			
Open source Systems									
JCM/Math Lib	924	0	0	757	0	167	18.07	6.02	16.01
InfinityPFM	165	0	0	133	0	32	19.39	3.67	7.87
Struts/WAF	100	0	0	79	0	21	21.00	0.89	3.61
ILedger	126	0	0	91	0	35	27.78	1.01	2.28
JasperReports/RE	1667	0	0	1493	0	174	10.44	2.29	8.42
Regatta	53	0	0	39	0	14	26.42	1.13	4.45
JSpider	63	0	0	56	0	7	11.11	1.65	6.45
JTidy	1203	0	0	1138	0	65	5.40	15.31	39.36
Total (Open Source)	8893	0	0	6701	0	2192	24.65	3.47	10.55
Industry Systems									
Blog	616	0	0	504	0	112	18.18	2.10	7.59
OfficeAssistant	132	0	0	79	0	53	40.15	0.77	2.10
ObjectLibrary	35	0	0	22	0	13	37.14	0.43	3.07
Redist	108	0	0	81	0	27	25.00	2.98	12.65
Total (Industry)	891	0	0	686	0	205	23.01	1.92	7.11
Error Seeded Systems									
JMeter/Load Test	389	0	0	337	0	52	13.37	1.86	9.34
XML-Security	366	0	0	305	0	61	16.67	6.23	13.78
Total (Error Seeded)	755	0	0	642	0	113	14.97	3.74	11.28
Grand Total	10539	0	0	8029	0	2510	23.82	3.17	6.99

Table A1.1.5. Results for the classification of additional infeasible paths detected by DIPLE from BP0 experiment on intra-paths

System	# InfP. detected by DIPLE due to replacement	# InfP. detected by DIPLE due to abstraction
Open source Systems		
JCM/Math Lib	167	0
InfinityPFM	32	0
Struts/WAF	21	0
ILedger	35	0
JasperReports/RE	174	0
Regatta	14	0
JSpider	7	0
JTidy	65	0
Total (Open Source)	515	0
Industry Systems		
Blog	112	0
OfficeAssistant	53	0
ObjectLibrary	13	0
Redist	27	0
Total (Industry)	205	0
Error Seeded Systems		
JMeter/Load Test	52	0
XML-Security	61	0
Total (Error Seeded)	113	0
Grand Total	833	0

Table A1.2.1. Size statistics of the BP0 experiment on inter-paths

System	Size (KLOC)	#RootProc	#Path	#Nodes/Path	
				Mean	Stdev
Open source Systems					
JCM/Math Lib	12.28	492	2363	52.17	64.97
InfinityPFM	6.91	459	938	33.18	51.15
Struts/WAF	18.46	2375	2814	19.72	26.39
ILedger	8.79	1214	1789	13.00	15.30
JasperReports/RE	106.63	6434	15747	22.92	29.91
Regatta	9.71	825	1808	22.64	17.47
JSpider	5.34	606	1099	20.81	19.38
JTidy	13.37	387	1578	45.68	69.60
Total (Open Source)	181.49	12792	28136	25.94	59.89
Industry Systems					
Blog	12.44	895	8004	27.76	27.20
OfficeAssistant	8.23	473	1804	11.64	16.79
ObjectLibrary	7.34	241	822	21.79	53.10
Redist	4.13	97	1041	23.13	20.70
Total (Industry)	32.14	1706	11671	24.48	33.84
Error Seeded Systems					
JMeter/Load Test	17.74	1168	2545	26.62	45.30
XML-Security	14.74	722	2184	24.40	24.29
Total (Error Seeded)	32.48	1890	4729	24.74	37.18
Grand Total	246.11	16388	44536	18.14	56.88

Table A1.2.2. Results of the BP0 experiment using exhaustive symbolic evaluation on inter-paths

System	#Path	exhaustive symbolic evaluation		
		#Path	#infP	#notInfP
Open Source Systems				
JCM/Math Lib	2363	843	726	117
InfinityPFM	938	186	145	41
Struts/WAF	2814	237	205	32
JLedger	1789	193	141	52
JasperReports/RE	15747	1549	1513	36
Tomcat/AS	17446	3816	3746	70
MoneyJar	604	50	44	6
Regatta	1808	261	249	12
JSpider	1099	223	107	116
JTidy	1578	1129	802	327
JWordInput	43	24	4	20
Total(Open Source)	28136	4621	3888	733
Industry Systems				
Blog	8004	1801	1264	537
OfficeAssistant	1804	677	15	662
ObjectLibrary	822	223	45	178
Redist	1041	609	108	501
Total(Industry)	11671	3310	1432	1878
Error Seeded Systems				
JMeter/Load Test	2545	537	431	106
XML-Security	2184	562	469	93
Total(Error Seeded)	4729	1099	900	199
Grand Total	44536	9030	6220	2810

Table A1.2.3. Results of the BP0 experiment using DIPLE on inter-paths

System	#Path	#infP Detected				Paths processed by Syntax Analysis only		% Pred Nodes Eval
		Pattern1	Pattern2	Sub-infP	Total	#	%	
Open source Systems								
JCM/Math Lib	2363	168	214	344	726	1257	53.20	36.10
InfinityPFM	938	56	59	30	145	684	72.92	40.13
Struts/WAF	2814	87	45	73	205	2480	88.13	42.24
JLedger	1789	47	49	45	141	1494	83.51	41.93
JasperReports/RE	15747	487	623	403	1513	12564	79.79	39.26
Regatta	1808	30	155	64	249	1382	76.44	31.73
JSpider	1099	60	14	33	107	927	84.35	39.91
JTidy	1578	193	364	245	802	625	39.51	45.39
JWordInput	43	1	2	1	4	36	83.72	46.67
Total (Open Source)	28136	1128	1523	1237	3888	21413	76.10	39.50
Industry Systems								
Blog	8004	656	239	369	1264	6379	79.70	29.05
OfficeAssistant	1804	7	1	7	15	1755	97.28	41.20
ObjectLibrary	822	11	16	18	45	605	73.60	40.22
Redis	1041	35	36	37	108	841	80.79	48.33
Total (Industry)	11671	709	292	431	1432	9580	82.08	33.43
Error Seeded Systems								
JMeter/Load Test	2545	229	111	91	431	1880	73.87	30.20
XML-Security	2184	149	66	254	469	1344	61.54	50.34
Total (Error Seeded)	4729	378	177	345	900	3224	68.18	39.50
Grand Total	44536	2215	1992	2013	6220	34217	76.83	38.39

Table A1.2.4. Results for comparing DIPLE with exhaustive symbolic evaluation from BP0 experiment on inter-paths

System	InfP. Detected							Avg Proc Time Per Path (ms)	
	DIPLE			exhaustive symbolic evaluation					
	#Correct	#False - Positive	#False - Negative	#Correct	#False - Positive	#False-Negative		DIPL E (ms)	exhaustiv e symbolic evaluation
						#	%		
Open source Systems									
JCM/Math Lib	726	0	0	566	0	160	22.04	12.45	23.27
InfinityPFM	145	0	0	118	0	27	18.62	6.68	20.66
Struts/WAF	205	0	0	133	0	72	35.12	2.78	8.58
ILedger [41]	141	0	0	111	0	30	21.28	3.55	10.79
JasperReports /RE	1513	0	0	972	0	541	35.76	5.46	18.06
Tomcat/AS	3746	0	0	3083	0	663	17.69	7.67	19.08
MoneyJar	44	0	0	30	0	14	31.81	3.91	14.04
Regatta	249	0	0	214	0	35	14.05	11.83	17.61
JSpider	107	0	0	73	0	34	31.78	2.22	12.38
JTidy	802	0	0	601	0	201	25.06	17.72	34.06
JWordInput	4	0	0	2	0	2	50.00	5.03	11.44
Total (Open Source)	7682	0	0	5903	0	1779	23.16	6.67	17.80
Industry Systems									
Blog	1264	0	0	1018	0	246	19.46	3.42	11.77
OfficeAssistant	15	0	0	12	0	3	20.00	2.05	4.75
ObjectLibrary	45	0	0	31	0	14	31.11	5.94	7.97
Redis	108	0	0	73	0	35	32.41	3.03	16.40
Total (Industry)	1432	0	0	1134	0	298	20.81	3.35	10.83
Error Seeded Systems									
JMeter/Load Test	431	0	0	391	0	40	9.28	2.95	10.57
XML-Security	469	0	0	389	0	80	17.08	4.57	15.41
Total (Error Seeded)	900	0	0	780	0	120	13.33	3.70	12.81
Grand Total	10014	0	0	7817	0	2197	21.94	3.35	15.46

Table A1.2.5. Results for the classification of additional infeasible paths detected by DIPLE from BP0 experiment on inter-paths

System	# InfP. detected by DIPLE due to replacement	# InfP. detected by DIPLE due to abstraction
Open source Systems		
JCM/Math Lib	154	6
InfinityPFM	27	0
Struts/WAF	72	0
JSLedger	30	0
JasperReports/RE	522	19
Regatta	35	0
JSpider	34	0
JTidy	201	0
Total (Open Source)	1075	25
Industry Systems		
Blog	246	0
OfficeAssistant	3	0
ObjectLibrary	14	0
Redis	35	0
Total (Industry)	298	0
Error Seeded Systems		
JMeter/Load Test	40	0
XML-Security	80	0
Total (Error Seeded)	120	0
Grand Total	1493	25

A2 – Results of CtP2 Experiment

Appendix A2 shows the tables and figures for the results of the CtP2 experiment.

Table A2.1.1. Size statistics of the CtP2 experiment on intra-paths

System	Size (KLOC)	#Proc	#Path	#Nodes/path	
				Mean	Stdev
Open source Systems					
ApacheEmail	1.28	139	17995	58.00	16.69
JSpider	5.34	1406	4875	18.49	19.54
JCapivara	23.95	3337	103391	56.06	29.47
Jampal	20.36	1169	57689	61.15	71.01
Total (Open Source)	50.93	6051	183950	56.85	41.22
Industry Systems					
CodeSystem	25.23	3489	61921	74.77	58.71
Hotel-Management	6.52	1162	6355	49.24	52.68
Total (Industry)	31.75	4651	68276	72.39	57.01
Error Seeded Systems					
XMLSecurity	14.74	1310	67060	64.00	33.51
Nano XML	3.26	438	9289	27.50	14.71
Total (Error Seeded)	18.00	1748	76349	59.56	31.73
Grand Total	100.68	12450	328575	36.64	52.47

Table A2.1.2. Results of the CtP2 experiment using exhaustive symbolic evaluation on intra-paths

System	#Path	exhaustive symbolic evaluation		
		#Path	#infP	#notInfP
Open Source Systems				
ApacheEmail	17995	10196	9830	366
Spider	4875	3098	516	2582
Capivara	103391	74389	21793	52596
ampal	57689	32190	6704	25486
Total (Open Source)	183950	119873	38843	81030
Industry Systems				
CodeSystem	61921	39778	21112	18666
Hotel-Management	6355	3297	619	2678
Total(Industry)	68276	43075	21731	21344
Error Seeded Systems				
XMLSecurity	67060	42867	28437	14430
Nano XML	9289	3043	2319	724
Total(Error Seeded)	76349	45910	30756	15154
Grand Total	328575	208858	91330	117528

Table A2.1.3. Results of the CtP2 experiment using DIPLE on intra-paths

System	#Path	#infP Detected				Paths processed by Syntax Analysis only		% Pred Nodes Eval
		Pattern1	Pattern2	Sub-infP	Total	#	%	
Open source Systems								
ApacheEmail	17995	498	1994	7338	9830	15227	84.62	21.30
Spider	4875	136	129	251	516	4155	85.23	17.33
Capivara	103391	2148	11071	8574	21793	81939	79.25	21.54
ampal	57689	2170	2711	1823	6704	42918	74.39	44.85
Total (Open Source)	183950	4952	15905	17986	38843	144239	78.41	28.72
Industry Systems								
CodeSystem	61921	2736	3832	14544	21112	50175	81.03	38.71
Hotel-Management	6355	215	134	270	619	5253	82.66	32.68
Total (Industry)	68276	2951	3966	14814	21731	55428	81.18	38.14
Error Seeded Systems								
XMLSecurity	67060	4386	6141	17910	28437	56214	83.83	34.05
Nano XML	9289	544	385	1390	2319	6519	70.18	29.18
Total (Error Seeded)	76349	4930	6526	19300	30756	62733	82.17	33.45
Grand Total	328575	12833	26397	52100	91330	262400	79.86	31.78

Table A2.1.4. Results for comparing DIPLE with exhaustive symbolic evaluation from CtP2 experiment on intra-paths

System	InfP Detected						Avg Proc Time Per Path (ms)		
	DIPLE			exhSymbEvaPhInf					
	#Correct	#False - Positive	#False - Negative	#Correct	#False - Positive	#False-Negative		DIPLE	exhaustive symbolic evaluation
						#	%		
Open source Systems									
apacheEmail	9830	0	0	7470	0	2360	24.01	19.25	27.03
Spider	516	0	0	319	0	197	38.18	2.42	10.94
Capivara	21793	0	0	17652	0	4141	19.00	6.88	17.92
ampal	6704	0	0	5296	0	1408	21.00	10.58	19.30
Total (Open Source)	38843	0	0	30737	0	8106	20.87	9.13	20.37
Industry Systems									
CodeSystem	21112	0	0	14567	0	6545	31.00	9.12	17.55
Hotel-Management	619	0	0	513	0	106	17.12	3.03	13.86
Total (Industry)	21731	0	0	15080	0	6651	30.61	8.55	17.44
Error Seeded Systems									
XMLSecurity	28437	0	0	19337	0	9100	32.00	6.11	21.77
Nano XML	2319	0	0	1646	0	673	29.02	4.15	11.66
Total (Error Seeded)	30756	0	0	20983	0	9773	31.78	5.87	21.01
Grand Total	91330	0	0	66800	0	24530	26.86	8.25	19.89

Table A2.1.5. Results for the classification of additional infeasible paths detected by DIPLE from CtP2 experiment on intra-paths

System	# InfP. detected by DIPLE due to replacement	# InfP. detected by DIPLE due to abstraction
Open source Systems		
ApacheEmail	2357	3
Spider	189	8
Capivara	4002	139
Jampal	1369	39
Total (Open Source)	7917	189
Industry Systems		
CodeSystem	6421	124
Hotel-Management	103	3
Total (Industry)	6524	127
Error Seeded Systems		
XMLSecurity	8743	357
Nano XML	663	10
Total (Error Seeded)	9406	367
Grand Total	23847	683

Table A2.2.1. Size statistics for the CtP2 experiment on inter-paths

System	Size (KLOC)	#RootProc	#Path	#Nodes/path	
				Mean	Stdev
Open source Systems					
ApacheEmail	1.28	73	37975	78.31	46.11
JSpider	5.34	606	8566	22.31	20.72
JCapivara	23.95	1289	231299	59.06	36.54
Jampal	20.36	389	203376	72.47	81.09
Total (Open Source)	50.93	2357	481216	65.59	67.48
Industry Systems					
CodeSystem	25.23	2040	72936	64.92	60.74
Hotel-Management	6.52	411	8753	45.20	52.59
Total (Industry)	31.75	2451	81689	62.81	59.03
Error Seeded Systems					
XMLSecurity	14.74	722	73148	76.89	52.07
Nano XML	3.26	189	57832	33.81	20.65
Total (Error Seeded)	18.00	911	130980	57.86	37.99
Grand Total	100.68	5719	693885	63.81	62.74

Table A2.2.2. Results of the Ctp2 experiment using exhaustive symbolic evaluation on inter-paths

System	#Path	exhaustive symbolic evaluation		
		#Path	#infP	#notInfP
Open Source Systems				
ApacheEmail	37975	24053	15731	8322
Spider	8566	5692	1217	4475
Capivara	231299	135333	45119	90214
ampal	203376	110230	39873	70357
Total (Open Source)	481216	275308	101940	173368
Industry Systems				
CodeSystem	72936	45301	27046	18255
Hotel-Management	8753	5413	1193	4220
Total(Industry)	81689	50714	28239	22475
Error Seeded Systems				
XMLSecurity	73148	50129	32511	17618
Nano XML	57832	26007	7934	18073
Total(Error Seeded)	130980	76136	40445	35691
Grand Total	693885	402158	170624	231534

Table A2.2.3. Results of the Ctp2 experiment using DIPLE on inter-paths

System	#Path	#infP Detected				Paths processed by Syntax Analysis only		% Pred Nodes Eval
		Pattern1	Pattern2	Sub-infP	Total	#	%	
Open source Systems								
ApacheEmail	37975	3179	5314	7238	15731	29985	78.96	33.41
JSpider	8566	241	329	647	1217	6944	81.06	29.85
JCapivara	231299	14587	14740	15792	45119	159689	69.04	36.07
Jampal	203376	7161	11121	21591	39873	145088	71.34	45.18
Total (Open Source)	481216	25168	31504	45268	101940	341706	71.01	39.60
Industry Systems								
CodeSystem	72936	6045	4839	16162	27046	53255	73.02	39.09
Hotel-Management	8753	318	539	336	1193	6216	71.01	34.72
Total (Industry)	81689	6363	5378	16498	28239	59471	72.80	38.62
Error Seeded Systems								
XMLSecurity	73148	6200	10423	15888	32511	54910	75.07	41.06
Nano XML	57832	2028	1457	4449	7934	39195	67.78	30.94
Total (Error Seeded)	130980	8228	11880	20337	40445	94105	71.85	36.60
Grand Total	693885	39759	48762	82103	170624	495282	71.38	38.92

Table A2.2.4. Results for comparing DIPLE with exhaustive symbolic evaluation from CtP2 experiment on inter-paths

System	InfP Detected							Avg Proc Time Per Path (ms)	
	DIPLE			exhSymbEvaPhInf					
	#Correct	#False - Positive	#False - Negative	#Correct	#False - Positive	#False -Negative		DIPLE	exhaustive symbolic evaluation
						#	%		
Open source Systems									
ApacheEmail	15731	0	0	12270	0	3461	22.00	15.71	30.03
ISpider	1217	0	0	949	0	268	22.02	2.47	14.13
ICapivara	45119	0	0	33388	0	11731	26.00	6.95	21.99
Iampal	39873	0	0	28309	0	11564	29.00	11.94	25.33
Total (Open Source)	101940	0	0	74916	0	27024	26.51	9.67	23.89
Industry Systems									
CodeSystem	27046	0	0	18661	0	8385	31.00	9.93	18.21
Hotel-Management	1193	0	0	823	0	370	31.01	4.42	20.81
Total (Industry)	28239	0	0	19484	0	8755	31.00	9.34	18.49
Error Seeded Systems									
XMLSecurity	32511	0	0	23733	0	8778	27.00	5.09	21.29
Nano XML	7934	0	0	6347	0	1587	20.00	6.15	11.44
Total (Error Seeded)	40445	0	0	30080	0	10365	25.63	5.56	16.94
Grand Total	170624	0	0	124480	0	46144	27.04	8.86	21.94

Table A2.2.5. Results for the classification of additional infeasible paths detected by DIPLE from CtP2 experiment on inter-paths

System	# InfP. detected by DIPLE due to replacement	# InfP. detected by DIPLE due to abstraction
Open source Systems		
ApacheEmail	3367	94
ISpider	258	10
ICapivara	11442	289
Iampal	11463	101
Total (Open Source)	26530	494
Industry Systems		
CodeSystem	8301	84
Hotel-Management	363	7
Total (Industry)	8664	91
Error Seeded Systems		
XMLSecurity	8778	0
Nano XML	1587	0
Total (Error Seeded)	10365	0
Grand Total	45559	585

A3 – Results of Random Experiment

Appendix A3 shows the tables and figures for the results of our Random experiment.

Table A3.1.1. Size statistics for the Random experiment on intra-paths

System	Size (KLOC)	#Node	#Edge	#Path	#Nodes/path	
					Mean	Stdev
Open source Systems						
DGraph	7.11	5420	5202	361	313.10	452.08
Phex/P2PC	61.07	43527	41789	2852	102.01	315.89
JtoPas/Java parser	2.96	2767	2721	79	76.32	168.38
Jhotdraw / GE	43.35	35877	35366	841	160.41	450.16
Total (Open Source)	114.49	87591	85078	4133	102.84	257.91
Industry Systems						
CodeSystem	25.23	19133	17142	3267	330.70	653.62
Hotel-Management	6.52	7773	7103	1101	174.42	363.53
Total (Industry)	31.75	26906	24245	4368	185.89	386.87
Error Seeded Systems						
Ant 1.8	78.82	35185	34344	1382	86.57	214.84
Nano XML	3.26	2755	2685	118	178.57	297.09
Total (Error Seeded)	82.08	37940	37029	1500	87.78	214.73
Grand Total	228.32	152437	146352	10001	136.88	333.27

Table A3.1.2. Results of the Random experiment using exhaustive symbolic evaluation on intra-paths

System	#Path	exhaustive symbolic evaluation		
		#Path	#infP	#notInfP
Open Source Systems				
DGraph	361	155	73	82
Phex/P2PC	2852	1220	310	910
JtoPas/Java parser	79	39	3	36
Jhotdraw / GE	841	272	80	192
Total (Open Source)	4133	1686	466	1220
Industry Systems				
CodeSystem	3267	1126	479	647
Hotel-Management	1101	381	69	312
Total(Industry)	4368	1507	548	959
Error Seeded Systems				
Ant 1.8	1382	580	176	404
Nano XML	118	62	27	35
Total(Error Seeded)	1500	642	203	439
Grand Total	10001	3835	1217	2618

Table A3.1.3 Results of the Random experiment using DIPLE on intra-paths

System	#Path	#infP Detected				Paths processed by Syntax Anlysis only		% Pred Nodes Eval
		Pattern1	Pattern2	Sub-infP	Total	#	%	
Open source Systems								
DGraph	361	24	26	23	73	192	53.19	25.93
Phex/P2PC	2852	130	103	77	310	2201	77.17	21.81
JtoPas/Java parser	79	2	1	0	3	71	89.87	19.62
Jhotdraw / GE	841	42	27	11	80	667	79.31	29.78
Total (Open Source)	4133	198	157	111	466	3131	75.76	23.75
Industry Systems								
CodeSystem	3267	123	146	210	479	2161	66.15	35.05
Hotel-Management	1101	23	19	27	69	946	85.92	27.87
Total (Industry)	4368	146	165	237	548	3107	71.13	33.24
Error Seeded Systems								
Ant 1.8	1382	59	96	21	176	1103	79.81	25.75
Nano XML	118	9	12	6	27	60	50.85	29.20
Total (Error Seeded)	1500	68	108	27	203	1163	77.53	26.02
Grand Total	10001	412	430	375	1217	7401	74.00	28.24

Table A3.1.4. Results for comparing DIPLE with exhaustive symbolic evaluation from Random experiment on intra-paths

System	InfP Detected							Avg Proc Time Per Path (ms)	
	DIPLE			exhSymbEvaPhInf					
	#Correct	#False - Positive	#False - Negative	#Correct	#False - Positive	#False-Negative		DIPLE	exhaustive symbolic evaluation
#						%			
Open source Systems									
DGraph	73	0	0	55	0	18	24.66	15.06	21.87
Phex/P2PC	310	0	0	213	0	97	31.29	22.46	34.04
JtoPas/Java parser	3	0	0	2	0	1	33.33	5.81	23.59
Jhotdraw / GE	80	0	0	62	0	18	22.50	10.17	28.62
Total (Open Source)	466	0	0	332	0	134	28.76	18.99	31.67
Industry Systems									
CodeSystem	479	0	0	335	0	144	30.06	22.46	39.51
Hotel-Management	69	0	0	52	0	17	24.64	7.01	22.57
Total (Industry)	548	0	0	387	0	161	29.38	18.57	35.24
Error Seeded Systems									
Ant 1.8	176	0	0	119	0	57	32.38	12.97	21.69
Nano XML	27	0	0	20	0	7	25.93	16.65	33.86
Total (Error Seeded)	203	0	0	139	0	64	31.53	13.26	55.55
Grand Total	1217	0	0	858	0	359	29.50	17.94	31.88

Table A3.1.5. Results for the classification of additional infeasible paths detected by DIPLE from Random experiment on intra-paths

System	# InfP. detected by DIPLE due to replacement	# InfP. detected by DIPLE due to abstraction
Open source Systems		
DGraph	17	1
Phex/P2PC	96	1
JtoPas/Java parser	1	0
Jhotdraw / GE	18	0
Total (Open Source)	132	2
Industry Systems		
CodeSystem	144	0
Hotel-Management	17	0
Total (Industry)	161	0
Error Seeded Systems		
Ant 1.8	55	2
Nano XML	7	0
Total (Error Seeded)	62	2
Grand Total	355	4

Table A3.2.1. Size statistics for the Random experiment on inter-paths

System	Size (KLOC)	#Node	#Edge	#Path	#Nodes/path	
					Mean	Stdev
Open source Systems						
DGraph	7.11	5420	5202	361	309.17	1464.62
Phex/P2PC	61.07	43527	41789	2852	155.34	525.78
JtoPas/Java parser	2.96	2767	2721	79	50.62	219.12
Jhotdraw / GE	43.35	35877	35366	841	445.53	1271.42
Total (Open Source)	114.48	87591	85078	4133	225.82	1187.51
Industry Systems						
CodeSystem	25.23	19133	17142	3267	269.11	956.81
Hotel-Management	6.52	7773	7103	1101	137.13	236.45
Total (Industry)	31.75	26906	24245	4368	235.84	766.17
Error Seeded Systems						
Ant 1.8	78.82	35185	34344	1382	145.33	451.70
Nano XML	3.26	2755	2685	118	195.86	991.72
Total (Error Seeded)	82.07	37940	37029	1500	149.31	508.33
Grand Total	228.32	152437	146352	10001	218.72	745.91

Table A3.2.2. Results of the Random experiment using exhaustive symbolic evaluation on inter-paths

System	#Path	exhaustive symbolic evaluation		
		#Path	#infP	#notInfP
Open Source Systems				
DGraph	361	31	23	8
Phex/P2PC	2852	1129	128	1001
JtoPas/Java parser	79	26	4	22
xJhotdraw / GE	841	225	19	206
Total (Open Source)	4133	1411	174	1237
Industry Systems				
CodeSystem	3267	461	59	402
Hotel-Management	1101	218	32	186
Total(Industry)	4368	679	91	588
Error Seeded Systems				
Ant 1.8	1382	786	80	706
Nano XML	118	25	15	10
Total(Error Seeded)	1500	811	95	716
Grand Total	10001	2901	360	2541

Table A3.2.3. Results of the Random experiment using DIPLE on inter-paths

System	#Path	#infP Detected				Paths processed by Syntax Anlysis only		% Pred Nodes Eval
		Pattern1	Pattern2	Sub-infP	Total	#	%	
Open source Systems								
DGraph	361	14	9	0	23	300	83.10	30.13
Phex/P2PC	2852	32	33	63	128	2559	89.73	51.00
JtoPas/Java parser	79	3	0	1	4	73	91.41	20.12
Jhotdraw / GE	841	14	3	2	19	796	94.65	30.13
Total (Open Source)	4133	63	45	66	174	3728	90.20	44.34
Industry Systems								
CodeSystem	3267	13	20	26	59	2409	73.74	20.86
Hotel-Management	1101	12	11	9	32	784	71.22	15.59
Total (Industry)	4368	25	31	35	91	3193	73.10	19.53
Error Seeded Systems								
Ant 1.8	1382	33	25	22	80	1207	87.34	17.10
Nano XML	118	6	8	1	15	98	83.05	44.95
Total (Error Seeded)	1500	39	33	23	95	1305	87.00	19.29
Grand Total	10001	127	109	124	360	8226	82.25	29.75

Table A3.2.4. Results for comparing DIPLE with exhaustive symbolic evaluation from Random experiment on inter-paths

System	InfP Detected							Avg Proc Time Per Path (ms)	
	DIPLE			exhaustive symbolic evaluation				DIPLE	exhaustive symbolic evaluation
	#Correct	#False - Positive	#False - Negative	#Correct	#False - Positive	#False- Negative # %			
Open source Systems									
DGraph	23	0	0	18	0	5	21.74	56.76	299.78
Phex/P2PC	128	0	0	103	0	25	19.53	22.15	29.67
JtoPas/Java parser	4	0	0	2	0	2	50.00	5.94	27.06
Jhotdraw / GE	19	0	0	15	0	4	21.05	36.17	118.56
Total (Open Source)	174	0	0	138	0	36	20.69	27.72	71.30
Industry Systems									
CodeSystem	59	0	0	46	0	13	22.03	46.93	295.87
Hotel-Management	32	0	0	25	0	7	21.88	21.28	90.84
Total (Industry)	91	0	0	71	0	20	21.98	40.47	244.19
Error Seeded Systems									
Ant 1.8	80	0	0	61	0	19	23.75	27.33	78.66
Nano XML	15	0	0	9	0	6	40.00	11.99	48.55
Total (Error Seeded)	95	0	0	70	0	25	26.31	26.12	76.29
Grand Total	360	0	0	279	0	81	22.50	33.05	118.09

Table A3.2.5. Results for the classification of additional infeasible paths detected by DIPLE from Random experiment on inter-paths

System	# InfP. detected by DIPLE due to replacement	# InfP. detected by DIPLE due to abstraction
Open source Systems		
DGraph	5	0
Phex/P2PC	24	1
JtoPas/Java parser	2	0
Jhotdraw / GE	4	0
Total (Open Source)	35	1
Industry Systems		
CodeSystem	13	0
Hotel-Management	7	0
Total (Industry)	20	0
Error Seeded Systems		
Ant 1.8	18	1
Nano XML	6	0
Total (Error Seeded)	24	1
Grand Total	79	2

APPENDIX B – NOTATION CONVENTIONS

Appendix B shows all the notation conventions used throughout this thesis.

Table B.1 General notations

Symbol	Meaning
$\emptyset$	An empty set
AST	Abstract Syntax Tree
CFG	Control flow graph
CodeSurfer	A C/C++ optimization framework
EIP	Extended instruction Pointer
FALSE	Boolean constant “false”
IIFG	Inter-procedural inline flow graph
SMT	Satisfiability Modulo Theories
SMT solver	Constraint solver to solve SMT instances
Soot	A Java optimization framework
SQL	Structured query language
SQL statement	A program statement that interacts with a database
TRUE	Boolean constant “true”
Z3	An SMT solver from Microsoft

Table B.2 Additional notations used by Chapter 4

Symbol/Variable	Meaning
$\Psi_{p, q}$	A set of weakly correlated predicate nodes between p and q in π , where p and q are correlated predicate nodes and π is a path.
π	A path
$\wp$	A set of flag-based predicate nodes used in <i>applyDiple</i>
$\wp_{\pi}$	A set of flag-based predicate nodes in π is extracted from the set $\wp$ of all flag-based predicate nodes in G
Γ	$\Gamma = \{(p, I_p) \mid p \text{ is a predicate node in } G \text{ and } I_p \text{ is the set of input variables that are transitively referenced at } p\}$, it is used in <i>applyDiple</i>
ξ	A set of infeasible sub-paths accumulated on the fly by <i>applyDiple</i>
$\mathcal{I}$	A set used by <i>applyDiple</i> to store identified infeasible paths
$\mathcal{R}$	The set of paired correlated predicate nodes in π
G	A control flow graph used in <i>applyDiple</i>
W	For each paired corrected predicate nodes(p, q) in $\mathcal{R}$, W is the corresponding $\Psi_{p, q}$
BP0 Experiment	Basis Path Coverage with 0 Loop Iteration Experiment
CtP2 Experiment	Ct Path Coverage with 2 Loop Iteration Experiment
exhSymbEvaPhInf	Exhaustive symbolic evaluation, used in an additional automated method to evaluate DIPLE’s performance.
l -constant propagation	Limited constant propagation
l -symbolic evaluation	Limited symbolic evaluation
<i>infeasibleSubPath</i>	The identified infeasible sub-path in π
<i>isInfeasible</i>	A boolean value indicating whether π is infeasible
<i>isInfPath</i>	An algorithm to identify whether a given path is infeasible
<i>lSymEvalPattern2</i>	An algorithm to perform l -symbolic evaluation

Table B.3. Additional notations used by Chapter 5

Symbol/Variable	Meaning
π	A path
ψ	A class of aliased variables along a path
ξ	The set of detected vulnerable paths
$\langle S_i, \beta_i, N_i \rangle$	If a path is vulnerable, S_i is the sink node, β_i is the corresponding path constraint, N_i is a set containing the <i>vulnerableSubRoutine</i> for S_i
<i>AliasAnalysis</i>	The proposed path sensitive alias analysis
<i>BOS node</i>	Buffer Overflow Sensitive node
<i>detectBufferOverflows</i>	The proposed approach to identify buffer overflow vulnerability
<i>evalPattern1</i>	The proposed algorithm to evaluate a path against pattern1
<i>evalPattern2</i>	The proposed algorithm to evaluate a path against pattern2
<i>evalPattern3</i>	The proposed algorithm to evaluate a path against pattern3
<i>vrn</i>	Vulnerable root node
<i>vulnerableSubRoutine</i>	Given a path, <i>vulnerableSubRoutine</i> is corresponding vulnerable sub path detected by the proposed method.

Table B.4 Additional notations used by Chapter 6

Symbol/Variable	Meaning
ABOR	Automatic Buffer Overflow Repairing, the proposed approach to remove buffer overflow vulnerability
bo-sensitive sink	Buffer overflow path sensitive sink

AUTHOR'S PUBLICATIONS

- [1] **S. Ding**, H.B.K. Tan, H.Y. Zhang, “Automatic Removal of Buffer Overflow Vulnerabilities in C/C++ Programs,” in Proceedings of *the 16th International Conference on Enterprise Information Systems (ICEIS)*, Lisbon, Portugal, 27-30 April, 2014. **Best Paper Award.**
- [2] **S. Ding**, H.Y. Zhang, H.B.K. Tan, “Detecting infeasible branches based on code patterns.”, in Proceedings of *the IEEE joint conference of the Working Conference on Reverse Engineering and the European Conference on Software Maintenance and Reengineering (CSMR-WCRE)*, Antwerp, Belgium, pp. 74-83, 3-7 February, 2014.
- [3] **S. Ding**, H.B.K. Tan, H.Y. Zhang, M.N. Ngo, “Detection of Infeasible Paths through Pattern Recognition”, (2nd round review) *Journal of Information and Software Technology*, 2014.
- [4] **S. Ding**, H.B.K. Tan, L.K. Shar, B.M. Padmanabhuni, “DIVV: A Hybrid Framework for Detecting Input Validation Vulnerabilities”, (submitted to) *IET Software*, 2014.
- [5] Y. Arnatovich, H.B.K. Tan, **S. Ding**, K.P. Liu and L.K. Shar, “Empirical Comparison of Intermediate Representations for Android Applications”, in Proceedings of *the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Hyatt Regency, Canada, 1-3 July, 2014.
- [6] Y. Cao, H.Y. Zhang, **S. Ding**, H.B.K. Tan, “SymCrash: Selective Recording for Reproducing Crashes”, in Proceedings of the *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Västerås, Sweden, 15-19 September, 2014.
- [7] L.B. Liu, **S. Ding**, “Optical coherence microscopy with extended depth of focus”, (submitted to) *SPIE Scanning Microscopies*, Monterey, California, United States, 16-18 September, 2014.
- [8] **S. Ding**, H.B.K. Tan, L.K. Shar, B.M. Padmanabhuni, “Towards a Hybrid Framework for Detecting Input Manipulation Vulnerabilities”, in Proceedings of *the 20th Asia-Pacific Software Engineering Conference (APSEC)*, Bangkok, Thailand, pp. 363-370, 2-5 December, 2013.
- [9] **S. Ding**, H.B.K. Tan, “Detection of Infeasible Paths: Approaches and Challenges”, *Springer Lecture Notes of Compute Science, Serial of Communications in Computer and Information Science (CCIS)*, Volume 410, pp 64-78, 2013.

- [10] **S. Ding**, H.B.K. Tan, “Survey of Detection of Infeasible Paths” in *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, Wroclaw, Poland, pp. 43-52, June 29-30, 2012. **Best Student Paper Award.**
- [11] **S. Ding**, H.B.K. Tan, Liu K.P., H.Y. Zhang, C. Mahinthan, “Detection of Buffer Overflow Vulnerabilities in C/C++ with Pattern based Limited Symbolic Evaluation,” in *Proceedings of the IEEE 36th International Conference on Computers, Software, and Applications (COMSAC)*, July 16-21, 2012, Izmir, Turkey, pp. 559-564.
- [12] **S. Ding**, H.B.K. Tan, “Design Verification for Database Applications through Analyzing their Support to Data Lifecycle”, *International Journal of Data Mining, Modelling and Management*, 2011, vol. 3, no. 4, pp. 406-417.
- [13] **S. Ding**, Q.P. Lin, L. Zhang, “Interactive and Collaborative Learning System”, *2nd International Conference on Ocean Engineering and Maritime Technology*, 1-3 April, 2009, Xiamen, China.
- [14] Q.P. Lin, L. Zhang, **S. Ding**, W.H. Wang, K.S. Neo, and G. Robert, “3D Collaborative Virtual Environment based e-learning system,” *International Journal of Applied Systemic Studies*, vol. 3, no. 2, 2010, pp. 211-226.
- [15] Q.P. Lin, L. Zhang, **S. Ding**, G. R. Feng, G.B. Huang, “Intelligent Mobile Agents for Large-Scale Collaborative Virtual Environment,” *International Journal of Virtual Reality*, 2008, vol. 7, no. 2, pp.63-72.

BIBLIOGRAPHY

- [1] *Control flow graph*. Available: [http://en.wikipedia.org/wiki/Dominator_\(graph_theory\)](http://en.wikipedia.org/wiki/Dominator_(graph_theory))
- [2] *CVE*. Available: <http://cve.mitre.org>
- [3] *Database Design*. Available: http://en.wikipedia.org/wiki/Database_design
- [4] *Mock Object*. Available: http://en.wikipedia.org/wiki/Mock_object
- [5] *Phex. File sharing tool*. Available: <http://www.phex.org/>
- [6] *Reference - C++ Reference*. Available: <http://www.cplusplus.com/reference/>
- [7] *Soot - a Java Optimization Framework*. Available: <http://www.sable.mcgill.ca/soot/>
- [8] *Sourceforge*. Available: <http://sourceforge.net/>
- [9] (2001). Exploiting Format String Vulnerabilities. Available: <https://www.cs.drexel.edu/~greenie/cs645/readings/formatstrings.pdf>
- [10] *Handbook of automated reasoning*. Elsevier Science Publishers B. V., 2001
- [11] (2001). *Java Components for Mathematics Project*. Available: <http://math.hws.edu/javamath/>
- [12] *Practical Object-Oriented Design With Uml*: McGraw-Hill Education (India) Pvt Limited, 2005.
- [13] (2012). *CodeSonar*. Available: <http://www.grammatech.com/products/codesonar>
- [14] (2012). *CodeSurfer*. Available: <http://www.grammatech.com/products/codesurfer>
- [15] (2012). *Satisfiability Modulo Theories (SMT)*. Available: http://en.wikipedia.org/wiki/Satisfiability_Modulo_Theories
- [16] (2012). *Software Design Pattern*. Available: http://en.wikipedia.org/wiki/Software_design_pattern
- [17] (2012). *Z3: SMT solver*. Available: <http://research.microsoft.com/en-us/um/redmond/projects/z3/ml/z3.html>
- [18] (2013). *Apache Tomcat and Struts*. . Available: <http://www.apache.org/>
- [19] (2013). *US-CERT*. Available: <http://www.us-cert.gov/>

- [20] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system for multiprocessing," presented at the Proceedings of the 1st International Conference on Supercomputing, Athens, Greece, 1988.
- [21] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, vol. 19, p. 137, 1976.
- [22] P. Altenbernd, "On the False Path Problem in Hard Real-Time Programs," in *Real-Time Systems, Euromicro Conference*, 1996, pp. 0102-0102.
- [23] A. P. Atlee and H. Gannon, "Specifying and Verifying Requirements of Real-Time Systems," *IEEE Trans. Softw. Eng.*, vol. 19, pp. 41-55, 1993.
- [24] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *Software, IEEE*, vol. 25, pp. 22-29, 2008.
- [25] N. Ayewah and W. Pugh, "Null dereference analysis in practice," presented at the Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, Toronto, Ontario, Canada, 2010.
- [26] G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, O. Wei, and A. Gupta, "SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement," presented at the Proceedings of the 15th international symposium on Static Analysis, Valencia, Spain, 2008.
- [27] T. Ball and S. K. Rajamani, "Bebop: a path-sensitive interprocedural dataflow engine," presented at the Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, Snowbird, Utah, United States, 2001.
- [28] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," *SIGPLAN Not.*, vol. 37, pp. 1-3, 2002.
- [29] M. Baluda, P. Braione, G. Denaro, and M. Pezze, "Structural coverage of feasible code," presented at the Proceedings of the 5th Workshop on Automation of Software Test, Cape Town, South Africa, 2010.
- [30] V. R. Basili, "The role of experimentation in software engineering: past, current, and future," presented at the Proceedings of the 18th international conference on Software engineering, Berlin, Germany, 1996.
- [31] V. R. Basili, F. Shull, and F. Lanubile, "Using Experiments to Build a Body of Knowledge," presented at the Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics, 2000.

- [32] W. G. Bently and E. F. Miller, "Ct coverage — initial results," *Software Quality Journal*, vol. 2, pp. 29-47, 1993/03/01 1993.
- [33] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," *SIGPLAN Not.*, vol. 47, pp. 453-462, 2012.
- [34] A. Bertolino and M. Marre, "Automatic generation of path covers based on the control flow analysis of computer programs," *Software Engineering, IEEE Transactions on*, vol. 20, pp. 885-899, 1994.
- [35] G. Bilardi and K. Pingali, "A framework for generalized control dependence," *SIGPLAN Not.*, vol. 31, pp. 291-300, 1996.
- [36] P. Bindu. (2011) Techniques for Defending from Buffer Overflow Vulnerability Security Exploits. Available: <http://doi.ieeecomputersociety.org/10.1109/MIC.2011.137>
- [37] P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks," presented at the Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Paris, France, 2008.
- [38] N. Bjørner, N. Tillmann, and A. Voronkov, "Path Feasibility Analysis for String-Manipulating Programs," presented at the Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, 2009.
- [39] R. Bodic, R. Gupta, and M. L. Soffa, "Refining data flow information using infeasible paths," *SIGSOFT Softw. Eng. Notes*, vol. 22, pp. 361-377, 1997.
- [40] R. Bodik, R. Gupta, and M. L. Soffa, "Interprocedural conditional branch elimination," *SIGPLAN Not.*, vol. 32, pp. 146-158, 1997.
- [41] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, *et al.*, "Automating structural testing of C programs: Experience with PathCrawler," in *Automation of Software Test, 2009. AST '09. ICSE Workshop on*, 2009, pp. 70-78.
- [42] P. Chalin, D. Sinnig, and K. Torkzadeh, "Capturing business transaction requirements in use case models," presented at the Proceedings of the 2008 ACM symposium on Applied computing, Fortaleza, Ceara, Brazil, 2008.
- [43] Q. Chen, L. Wang, and Z. Yang, "SAM: self-adaptive dynamic analysis for multithreaded programs," presented at the Proceedings of the 7th international Haifa

- Verification conference on Hardware and Software: verification and testing, Haifa, Israel, 2012.
- [44] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: algorithmic verification and debugging," *Commun. ACM*, vol. 52, pp. 74-84, 2009.
 - [45] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*: MIT Press, 2000.
 - [46] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil, "The right algorithm at the right time: comparing data flow analysis algorithms for finite state verification," presented at the Proceedings of the 23rd International Conference on Software Engineering, Toronto, Ontario, Canada, 2001.
 - [47] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-experimentation: Design & analysis issues for field settings*: Houghton Mifflin Boston, 1979.
 - [48] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, and H. Zheng, "Bandera: extracting finite-state models from Java source code," presented at the Proceedings of the 22nd international conference on Software engineering, Limerick, Ireland, 2000.
 - [49] A. L. Correa and C. M. L. Werner, "Precise Specification and Validation of Transactional Business Software," presented at the Proceedings of the Requirements Engineering Conference, 12th IEEE International, 2004.
 - [50] P. Cousot and R. Cousot, "Modular Static Program Analysis," presented at the Proceedings of the 11th International Conference on Compiler Construction, 2002.
 - [51] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static Detection of Vulnerabilities in x86 Executables," in *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, 2006, pp. 269-278.
 - [52] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," in *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, 2003, pp. 227-237.
 - [53] F. Cristian, "Correct and Robust Programs," *IEEE Trans. Softw. Eng.*, vol. 10, pp. 163-174, 1984.
 - [54] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: a safe execution environment for commodity operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 351-366, 2007.

- [55] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," Brown University, 1991.
- [56] R. Cytron, J. Ferrante, and V. Sarkar, "Compact representations for control dependence," *SIGPLAN Not.*, vol. 25, pp. 337-351, 1990.
- [57] M. Das, S. Lerner, and M. Seigle, "ESP: path-sensitive program verification in polynomial time," *SIGPLAN Not.*, vol. 37, pp. 57-68, 2002.
- [58] M. Delahaye, B. Botella, and A. Gotlieb, "Explanation-Based Generalization of Infeasible Path," presented at the Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, 2010.
- [59] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," presented at the Proceedings of the 28th international conference on Software engineering, Shanghai, China, 2006.
- [60] S. Ding. (2014). *Research on Program Analysis*. Available: <http://www.sunshine-nanyang.com/>
- [61] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Softw. Engg.*, vol. 10, pp. 405-435, 2005.
- [62] B. Dobing and J. Parsons, "Understanding the role of use cases in UML: A review and research agenda," *Journal of Database Management*, vol. 11, pp. 28-36, 2000.
- [63] N. Dor, S. Adams, M. Das, and Z. Yang, "Software validation via scalable path-sensitive value flow analysis," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 12-22, 2004.
- [64] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: towards a realistic tool for statically detecting all buffer overflows in C," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, San Diego, California, USA, 2003, pp. 155-167.
- [65] E. Duesterwald, R. Gupta, and M. L. Soffa, "Demand-driven computation of interprocedural data flow," presented at the Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Francisco, California, USA, 1995.
- [66] E. Duesterwald, R. Gupta, and M. L. Soffa, "A demand-driven analyzer for data flow testing at the integration level," presented at the Proceedings of the 18th international conference on Software engineering, Berlin, Germany, 1996.

- [67] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich, "Flow analysis for verifying properties of concurrent software systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 13, pp. 359-430, 2004.
- [68] A. Einarsson and J. D. Nielsen, "A Survivor's Guide to Java Program Analysis with Soot," *Notes from Department of Computer Science*, 2008.
- [69] A. Ermedahl, *A Modular Tool Architecture for Worst-Case ExecutionTime Analysis*: VDM Verlag, 2008.
- [70] A. Fehnker, R. Huuck, S. Seefried, and M. Tapp, "Fade to Grey: Tuning Static Program Analysis," *Electron. Notes Theor. Comput. Sci.*, vol. 266, pp. 17-32, 2010.
- [71] D. Feitelson. (2007) Experimental computer science: Introduction. *Communication of the ACM*. 24-26.
- [72] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319-349, 1987.
- [73] J. Fischer, R. Jhala, and R. Majumdar, "Joining dataflow with predicates," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 227-236, 2005.
- [74] C. Flanagan and S. Qadeer, "Predicate abstraction for software verification," *SIGPLAN Not.*, vol. 37, pp. 191-202, 2002.
- [75] I. Forgacs and A. Bertolino, "Feasible test path selection by principal slicing," Computer and Automation Research Institute of the Hungarian Academy of Sciences 1997.
- [76] M. Fowler, *Patterns of Enterprise Application Architecture*: Addison-Wesley Longman Publishing Co., Inc., 2002.
- [77] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *Software Engineering, IEEE Transactions on*, vol. 14, pp. 1483-1498, 1988.
- [78] E. Freeman, E. Freeman, B. Bates, and K. Sierra, *Head First Design Patterns*: O'Reilly Media Associates, Inc., 2004.
- [79] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil, "On Two Problems in the Generation of Program Test Paths," *IEEE Trans. Softw. Eng.*, vol. 2, pp. 227-231, 1976.
- [80] K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 751-761, 1991.

- [81] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [82] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek, "Buffer overrun detection using linear programming and static analysis," presented at the Proceedings of the 10th ACM conference on Computer and communications security, Washington D.C., USA, 2003.
- [83] N. Gehani, *The Database Book: Principles & Practice Using MySQL*: Silicon Press, 2011.
- [84] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," presented at the Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis, Seattle, Washington, United States, 1994.
- [85] J. Gustaffson, A. Ermedahl, and B. Lisper, "Algorithms for Infeasible Path Calculation. ," presented at the Sixth International Workshop on Worst-Case Execution Time Analysis, Dagstuhl, Germany, 2006.
- [86] J. Gustafsson, *Analyzing Execution-time of Object-oriented Programs Using Abstract Interpretation*: Institutionen för datorteknik, Univ., 2000.
- [87] J. Gustafsson, "Eliminating annotations by automatic flow analysis of real-time programs," presented at the Proceedings of the Seventh International Conference on Real-Time Systems and Applications, 2000.
- [88] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, 2006, pp. 57-66.
- [89] B. Hackett, M. Das, D. Wang, and Z. Yang, "Modular checking for buffer overflows in the large," presented at the Proceedings of the 28th international conference on Software engineering, Shanghai, China, 2006.
- [90] M. Hafiz and R. Johnson, "A security oriented program transformation to "add on" policies to prevent injection attacks," presented at the Proceedings of the 2nd Workshop on Refactoring Tools, Nashville, Tennessee, 2008.
- [91] M. J. Harrold, G. Rothermel, and S. Sinha, "Computation of interprocedural control dependence," *SIGSOFT Softw. Eng. Notes*, vol. 23, pp. 11-20, 1998.

- [92] D. Hedley and M. A. Hennell, "The causes and effects of infeasible paths in computer programs," presented at the Proceedings of the 8th international conference on Software engineering, London, England, 1985.
- [93] G. J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, pp. 279-295, 1997.
- [94] G. J. Holzmann and M. H. Smith, "An Automated Verification Method for Distributed Systems Software Based on Model Extraction," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 364-377, 2002.
- [95] S. Horwitz, T. Reps, and M. Sagiv, "Demand interprocedural dataflow analysis," *SIGSOFT Softw. Eng. Notes*, vol. 20, pp. 104-115, 1995.
- [96] J. Huang, "Detection of data flow anomaly through program instrumentation," *IEEE Trans. Software Eng.*, vol. 5, pp. 226-236, 1979.
- [97] I. Jacobson, *The Unified Software Development Process*: Pearson Education, 1999.
- [98] G. Jan, "Worst Case Execution Time Analysis of Object-Oriented Programs," 2002, pp. 0071-0071.
- [99] R. Jhala and R. Majumdar, "Interprocedural analysis of asynchronous programs," *SIGPLAN Not.*, vol. 42, pp. 339-350, 2007.
- [100] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," presented at the Proceedings of the 16th international conference on World Wide Web, Banff, Alberta, Canada, 2007.
- [101] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: Server-Side Detection of Cross-Site Scripting Attacks," presented at the Proceedings of the 2008 Annual Computer Security Applications Conference, 2008.
- [102] N. D. Jones, "Flow Analysis of Lambda Expressions" presented at the Proceedings of the 8th Colloquium on Automata, Languages and Programming, 1981.
- [103] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," *SIGPLAN Not.*, vol. 44, pp. 110-120, 2009.
- [104] U. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*: CRC Press, Inc., 2009.
- [105] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL Injection and cross-site scripting attacks," presented at the Proceedings of the 31st International Conference on Software Engineering, 2009.

- [106] G. A. Kildall, "A unified approach to global program optimization," presented at the Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Boston, Massachusetts, 1973.
- [107] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385-394, 1976.
- [108] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, *et al.*, "Preliminary guidelines for empirical research in software engineering," *Software Engineering, IEEE Transactions on*, vol. 28, pp. 721-734, 2002.
- [109] B. Korel, "Automated test data generation for programs with procedures," *SIGSOFT Softw. Eng. Notes*, vol. 21, pp. 209-215, 1996.
- [110] A. Kundu and E. Bertino, "A New Class of Buffer Overflow Attacks," presented at the Proceedings of the 2011 31st International Conference on Distributed Computing Systems, 2011.
- [111] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, 2011.
- [112] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," presented at the Proceedings of the 10th conference on USENIX Security Symposium - Volume 10, Washington, D.C., 2001.
- [113] W. Le and M. L. Soffa, "Marple: a demand-driven path-sensitive buffer overflow detector," presented at the Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, Atlanta, Georgia, 2008.
- [114] W. Le and M. L. Soffa, "Marple: Detecting faults in path segments using automatically generated analyses," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, p. 18, 2013.
- [115] W. Lei, Z. Qiang, and Z. Pengchao, "Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking," in *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, 2008, pp. 165-173.
- [116] K.-S. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Softw. Pract. Exper.*, vol. 33, pp. 423-460, 2003.
- [117] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "AutoPaG: towards automated software patch generation with source code root cause identification and repair," presented at

the Proceedings of the 2nd ACM symposium on Information, computer and communications security, Singapore, 2007.

- [118] H. Liu and H. B. K. Tan, "An Approach to Aid the Understanding and Maintenance of Input Validation," presented at the Proceedings of the 22nd IEEE International Conference on Software Maintenance, 2006.
- [119] H. Liu and H. B. K. Tan, "An approach for the maintenance of input validation," *Inf. Softw. Technol.*, vol. 50, pp. 449-461, 2008.
- [120] H. Liu and H. B. K. Tan, "Testing input validation in Web applications through automated model recovery," *J. Syst. Softw.*, vol. 81, pp. 222-233, 2008.
- [121] H. Liu and H. B. K. Tan, "Covering code behavior on input validation in functional testing," *Inf. Softw. Technol.*, vol. 51, pp. 546-553, 2009.
- [122] J. P. Loyall and S. A. Mathisen, "Using Dependence Analysis to Support the Software Maintenance Process," presented at the Proceedings of the Conference on Software Maintenance, 1993.
- [123] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.
- [124] A. Majumdar and C. Thomborson, "Manufacturing opaque predicates in distributed systems for code obfuscation," presented at the Proceedings of the 29th Australasian Computer Science Conference - Volume 48, Hobart, Australia, 2006.
- [125] N. Malevris, D. F. Yates, and A. Veevers, "Predictive metric for likely feasibility of program paths," *J. Electron. Mater.*, vol. 19, pp. 115-118, 1990.
- [126] M. Mäntylä and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?," *Software Engineering, IEEE Transactions on*, vol. 35, pp. 430-448, 2009.
- [127] C. C. McGeoch, *A Guide to Experimental Algorithmics*: Cambridge University Press, 2012.
- [128] J. Midtgaard, "Control-flow analysis of functional programs," *ACM Comput. Surv.*, vol. 44, pp. 1-33, 2012.
- [129] T. C. Miller and T. d. Raadt, "strncpy and strlcat: consistent, safe, string copy and concatenation," presented at the Proceedings of the annual conference on USENIX Annual Technical Conference, Monterey, California, 1999.
- [130] L. D. Moura and N. Bjorner, "Z3: an efficient SMT solver," presented at the Proceedings of the Theory and practice of software, 14th international conference on

Tools and algorithms for the construction and analysis of systems, Budapest, Hungary, 2008.

- [131] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*: Wiley Publishing, 2011.
- [132] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for c," *SIGPLAN Not.*, vol. 44, pp. 245-258, 2009.
- [133] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, pp. 477-526, 2005.
- [134] M. N. NGO, "Applying Program Analysis Augmented with Empirical Properties into Software Analysis, Testing and Maintenance," Doctor of Philosophy, School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore, 2009.
- [135] M. N. Ngo, "Applying program analysis augmented with empirical properties into software analysis, testing and maintenance," PHD Thesis, School of Electrical & Electronic Engineering, Nanyang Technological University, Singapore, 2009.
- [136] M. N. Ngo and H. B. K. Tan, "Detecting large number of infeasible paths through recognizing their patterns," presented at the Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, Dubrovnik, Croatia, 2007.
- [137] M. N. Ngo and H. B. K. Tan, "Applying static analysis for automated extraction of database interactions in web applications," *Inf. Softw. Technol.*, vol. 50, pp. 160-175, 2008.
- [138] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," *Inf. Softw. Technol.*, vol. 50, pp. 641-655, 2008.
- [139] M. Ochodek and J. Nawrocki, "Automatic Transactions Identification in Use Cases," in *Balancing Agility and Formalism in Software Engineering*. vol. 5082, B. Meyer, J. Nawrocki, and B. Walter, Eds., ed: Springer Berlin Heidelberg, 2008, pp. 55-68.
- [140] A. One. (1996), Smashing The Stack For Fun And Profit. Available: http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
- [141] R. Osherove, *The Art of Unit Testing: With Examples in .Net*: Manning Publications Co., 2009.

- [142] H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote, "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address," *Computers, IEEE Transactions on*, vol. 55, pp. 1271-1285, 2006.
- [143] B. Padmanabhuni and H. B. K. Tan, "Defending against Buffer-Overflow Vulnerabilities," *Computer*, vol. 44, pp. 53-60, 2011.
- [144] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: a roadmap," presented at the Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, 2000.
- [145] J. Poole. (1995), A Method to Determine a Basis Set of Paths to Perform Program Testing. Available: <http://hissa.nist.gov/publications/nistir5737/>
- [146] R. E. Prather and J. J. P. Myers, "The Path Prefix Software Testing Strategy," *IEEE Trans. Softw. Eng.*, vol. 13, pp. 761-766, 1987.
- [147] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," presented at the Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Francisco, California, USA, 1995.
- [148] G. Robert, M. Ulrich, and W. Reinhard, "Invariance of Approximative Semantics with Respect to Program Transformations."
- [149] J. R. Ruthruff, S. Elbaum, and G. Rothermel, "Experimental program analysis: a new program analysis paradigm," in *Proceedings of the 2006 international symposium on Software testing and analysis*, 2006, pp. 49-60.
- [150] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," presented at the Proceedings of the eighteenth international symposium on Software testing and analysis, Chicago, IL, USA, 2009.
- [151] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 263-272, 2005.
- [152] L. K. Shar and H. B. K. Tan, "Automated removal of cross site scripting vulnerabilities in web applications," *Inf. Softw. Technol.*, vol. 54, pp. 467-478, 2012.
- [153] J. Sifakis, "A vision for computer science — the system perspective," *Central European Journal of Computer Science*, vol. 1, pp. 108-116, 2011/03/01 2011.
- [154] S. Sinha and M. J. Harrold, "Analysis and Testing of Programs with Exception Handling Constructs," *IEEE Trans. Softw. Eng.*, vol. 26, pp. 849-871, 2000.

- [155] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, pp. 209-254, 2001.
- [156] D. I. K. Sjöberg, T. Dyba, and M. Jorgensen, "The Future of Empirical Methods in Software Engineering Research," presented at the 2007 Future of Software Engineering, 2007.
- [157] H. Tahbaldar and B. Kalita, "Automated Software Test Data Generation: Direction of Research," *International Journal of Computer Science and Engineering Survey*, vol. 2, pp. 99-120, 2011.
- [158] K. Taneja, Y. Zhang, and T. Xie, "MODA: automated test generation for database applications via mock objects," presented at the Proceedings of the IEEE/ACM international conference on Automated software engineering, Antwerp, Belgium, 2010.
- [159] X. Tao, N. Tillmann, J. De Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Dependable Systems & Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, 2009, pp. 359-368.
- [160] T. J. Teorey, S. Buxton, L. Fryman, R. H. Gting, T. Halpin, J. L. Harrington, *et al.*, *Database Design: Know It All*: Morgan Kaufmann Publishers Inc., 2008.
- [161] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz, "Experimental evaluation in computer science: a quantitative study," *J. Syst. Softw.*, vol. 28, pp. 9-18, 1995.
- [162] N. Tillmann and J. D. Halleux, "Pex: white box test generation for .NET," presented at the Proceedings of the 2nd international conference on Tests and proofs, Prato, Italy, 2008.
- [163] M. Vallentin. (2007), On the Evolution of Buffer Overflows. Available: http://matthias.vallentin.net/course-work/buffer_overflows.pdf
- [164] S. Vergilio, J. Maldonado, and M. Jino, "Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction," *Journal of the Brazilian Computer Society*, vol. 12, pp. 73-88, 2006/02/01 2006.
- [165] W. Visser, K. Havelund, G. Brat, and S. Park, "Java Pathfinder - Second Generation of a Java Model Checker," ed, 2000.
- [166] A. Vogelsang, A. Fehnker, R. Huuck, and W. Reif, "Software metrics in static program analysis," presented at the Proceedings of the 12th international conference on Formal engineering methods and software engineering, Shanghai, China, 2010.

- [167] D. Wagner, J. Foster, E. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," in *Network and Distributed System Security Symposium*, ed, 2000, pp. 3-17.
- [168] A. H. Watson and T. J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," 1982.
- [169] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 181-210, 1991.
- [170] E. J. Weyuker, "The cost of data flow testing: an empirical study," *Software Engineering, IEEE Transactions on*, vol. 16, pp. 121-128, 1990.
- [171] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," in *Network and Distributed System Security Symposium(NDSS)*, 2003, pp. 149-162.
- [172] N. Williams, "Abstract path testing with PathCrawler," presented at the Proceedings of the 5th Workshop on Automation of Software Test, Cape Town, South Africa, 2010.
- [173] N. Williams, B. Marre, P. Mouy, and M. Roger, "PathCrawler: automatic generation of path tests by combining static and dynamic analysis," presented at the Proceedings of the 5th European conference on Dependable Computing, Budapest, Hungary, 2005.
- [174] Y. Xie, A. Chou, and D. Engler, "ARCHER: using symbolic, path-sensitive analysis to detect memory access errors," in *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, Helsinki, Finland, 2003, pp. 327-336.
- [175] J. Xu, Z. Kalbarczyk, S. Patel, and I. Ravishankar, "Architecture Support for Defending Against Buffer Overflow Attacks," in *Second Workshop on Evaluating and Architecting System dependability*, 2002, pp. 55-62.
- [176] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against C and C++ programs," *ACM Comput. Surv.*, vol. 44, pp. 1-28, 2012.
- [177] M. V. Zelkowitz and D. R. Wallace, "Experimental Models for Validating Technology," *Computer*, vol. 31, pp. 23-31, 1998.
- [178] J. Zhang, "Symbolic Execution of Program Paths Involving Pointer and Structure Variables," presented at the Proceedings of the Quality Software, Fourth International Conference, 2004.

- [179] J. Zhang and X. Wang, "A Constraint Solver and Its Application to Path Feasibility Analysis," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, pp. 139-156, 2001.
- [180] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, pp. 366-427, 1997.
- [181] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 97-106, 2004.
- [182] S. Ding, H.B.K. Tan, H.Y. Zhang, "Automatic Removal of Buffer Overflow Vulnerabilities in C/C++ Programs," in Proceedings of the 16th International Conference on Enterprise Information Systems (ICEIS), Lisbon, Portugal, 27-30 April, 2014.
- [183] S. Ding, H.Y. Zhang, H.B.K. Tan, "Detecting infeasible branches based on code patterns," in Proceedings of the IEEE joint conference of the Working Conference on Reverse Engineering and the European Conference on Software Maintenance and Reengineering (CSMR-WCRE), Antwerp, Belgium, pp. 74-83, 3-7 February, 2014.
- [184] S. Ding, H.B.K. Tan, H.Y. Zhang, M.N. Ngo, "Detection of Infeasible Paths through Pattern Recognition", (2nd round review) *Journal of Information and Software Technology*, 2014.
- [185] S. Ding, H.B.K. Tan, L.K. Shar, B.M. Padmanabhuni, "DIVV: A Hybrid Framework for Detecting Input Validation Vulnerabilities", (submitted to) *IET Software*, 2014.
- [186] Y. Arnatovich, H.B.K. Tan, S. Ding, K.P. Liu and L.K. Shar, "Empirical Comparison of Intermediate Representations for Android Applications", in Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE), Hyatt Regency, Canada, 1-3 July, 2014.
- [187] Y. Cao, H.Y. Zhang, S. Ding, H.B.K. Tan, "SymCrash: Selective Recording for Reproducing Crashes", in Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE), Västerås, Sweden, 15-19 September, 2014.
- [188] S. Ding, H.B.K. Tan, L.K. Shar, B.M. Padmanabhuni, "Towards a Hybrid Framework for Detecting Input Manipulation Vulnerabilities", in the Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC), Bangkok, Thailand, pp. 363-370, 2-5 December, 2013.

- [189] S. Ding, H.B.K. Tan, “Detection of Infeasible Paths: Approaches and Challenges”, *Springer Lecture Notes of Compute Science, Serial of Communications in Computer and Information Science (CCIS)*, Volume 410, pp 64-78, 2013.
- [190] S. Ding, H.B.K. Tan, “Survey of Detection of Infeasible Paths”, in *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, Wroclaw, Poland, pp. 43-52, June 29-30, 2012.
- [191] S. Ding, H.B.K. Tan, Liu K.P., H.Y. Zhang, C. Mahinthan, “Detection of Buffer Overflow Vulnerabilities in C/C++ with Pattern based Limited Symbolic Evaluation”, in *Proceedings of the IEEE 36th International Conference on Computers, Software, and Applications (COMSAC)*, July 16-21, 2012, Izmir, Turkey, pp. 559-564.
- [192] S. Ding, H.B.K. Tan, “Design Verification for Database Applications through Analyzing their Support to Data Lifecycle”, *International Journal of Data Mining, Modelling and Management*, 2011, vol. 3, no. 4, pp. 406-417.