Goal-Oriented Dynamic Test Generation

A thesis submitted to

Auckland University of Technology (AUT)

in fulfilment of the requirements for the degree of

Doctor of Philosophy (PhD)

TheAnh Do

2013

School of Computing and Mathematical Sciences

Primary Supervisor: Prof. Alvis Fong

Table of Contents

List of Figures

List of Tables

Attestation of Authorship

"I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another author (expect where explicitly defined in the acknowledgements), nor material which to a substantial extension has been submitted to the award of any other degree or diploma of a university or other institute of higher learning."

Signed: 

Acknowledgements

I would never have accomplished this dissertation without the love and support of many people.

First and foremost, I am deeply grateful to my supervisors, Professor Alvis Cheuk Ming Fong and Dr Russel Pears. Professor Alvis has been with me on every step of my PhD journey. He knew me when I was in Viet Nam, helped me form the research proposal, and patiently supported me to find a scholarship to pursue a PhD degree in New Zealand. From Professor Alvis I received his insightful suggestions, hearty support, and invaluable encouragement during the course of my studies. His research methods and the wisdom he shared with me in our meetings were important contributions toward the completion of my PhD. From Dr Russel I received careful guidance and motivation to conduct my research. I am indebted to you both.

I am thankful to numerous staff at the AUT School of Computing and Mathematical Sciences (SCMS), as well as colleagues in the PhD Lab and the Software Engineering Research Lab (SERL) for their friendship and advice, their moral and social support, and for teaching and sharing their knowledge and research topics. Anne Philpott and Dr Stefan Marks suggested I become a teaching assistant in SCMS where I had the chance to socialize and interact with students. The PhD Lab has been my second home during the last three years and I found fabulous friends whose incredible hard work, determination, and perseverance helped to keep me motivated. SERL has been an amazing research group for me to familiarize myself with others' research, and to learn more about industrial practices.

I am also indebted to SCMS for offering me the three-year full-time scholarship so that I could concentrate on my research.

I am grateful to Professor Thoai Nam, Professor Quan Thanh Tho, and Dr Bui Hoai Thang for their constant support and guidance when I was working at Ho Chi Minh City University of Technology, Viet Nam. Thanks to these lovely people, I could prepare myself well before I commenced my PhD study in New Zealand. In New Zealand, I am particularly thankful to Hoang Thi Dung for her enduring friendship and support during the course of my study.

And finally, I am heartily grateful to my family, Mom, Dad, and my younger brother, who were always there to support me at all times, no matter what I did. I love you all!

Abstract

Automated software testing is increasingly being seen as an important means of improving the quality and reliability of software in industry. It mitigates the hardship of manual testing, which is labour-intensive and error-prone, and alleviates the expensive cost of software testing, which often accounts for around half of total software development costs. One way of enhancing automated software testing is to automate the process of test input generation. Over the last three decades, a considerable research effort has attempted to achieve this goal. This thesis concentrates on the scalability problem of the test input generation process, which lies at the heart of the automation of the software testing process. It develops techniques to perform test input generation in a goal-oriented mechanism in order to achieve high structural coverage criteria and maximize security vulnerability detection. The techniques developed in this thesis are based on well-established theoretical foundations of program analysis and software testing. They distinguish themselves from existing techniques through their capability to precisely identify a root cause leading to the execution of a specific test goal and to perform test input generation in a directed automated manner toward effectively and efficiently exploring the test goal.

A comparative evaluation was conducted via two sets of experiments in which our proposed techniques significantly outperformed existing techniques. Specifically, on a benchmark of 15 simulated and real world test subjects, our structural coverage testing technique significantly optimized the test input generation effort to achieve the highest structural coverage when compared to state-of-the-art techniques. Additionally, on a benchmark of 23 buffer overflow vulnerabilities, our security testing technique discovered security vulnerability defects within a matter of a few seconds, while existing techniques failed even after 30 minutes of testing on a number of test subjects.

This thesis contributes to scientific knowledge by enriching the application of computer science theory, and proposes techniques to improve the efficiency of automated software testing.

Chapter 1

Introduction

1.1 Motivation

Human reliance on the functioning of software systems is growing rapidly. Software systems are constantly becoming larger, more complex, and are continuously evolving. Ensuring the high quality and reliability of these systems is therefore an ongoing challenge. In fact, the United States' National Institute of Standards and Technology (NIST) estimated in 2002 that software failures cost the US economy alone about $59.5 billion (0.6% of GDP) every year, and that improvements in software testing infrastructure might save one-third of this cost [98]. Research reports and the media continuously highlight the impact of software disasters that cause economic losses and social problems, and can even cost human lives [41], [57], [96]. These realities drive the demand for developing effective and efficient techniques to assure software to be reliable, robust, safe, and secure.

Among the various kinds of techniques proposed and used, peer reviewing [74], [134] and software testing [4], [108] are the major software verification techniques adopted widely in the industrial practice of software development. While a peer review amounts to a software inspection performed by a team of software engineers to analyse and catch defects completely statically on the uncompiled code, software testing is the process of

actually executing the compiled code under consideration with test cases to compare the actual output to the desired output inferred from the software specification. Peer review involves a range of dedicated types of peer review procedures for specific error-detection goals. This technique is generally able to catch between 31% and 93% of the defects with a median of around 60% in practice [10]. However, the almost completely manual nature of peer review prevents it from exposing subtle errors such as algorithm defects and security breaches.

In software development, testing constitutes a significant part of any software engineering project. Up to 50% of the total software development costs are devoted to testing [108]. The major advantage of testing is that it is applicable to all kinds of software, ranging from application software to compilers and operating systems. Unfortunately, the process of testing software is labour-intensive and error-prone. The limitations of software testing, which apply throughout the entire process, can be usefully categorized as follows [4], [7], [133]:

- First, test inputs are hand-constructed. Manually constructing test inputs is tiresome, expensive, and unreliable. Automated test input generation techniques such as random testing and symbolic execution are limited in many aspects of real world software systems.
- Second, software testing is ad hoc. This is due to a lack of sufficient formal foundations to specify software requirements and to formulate specifications in the code. In practice, the translation of the specification to program assertions is mostly done manually.
- Third, the state space of the program can be huge, often infinite. Exhaustively checking all the possible program states is impractical. Hence, often software testing ends up with small portions of the state space tested, leaving unknowns about the untested space. This phenomenon is emphasized in the most often cited aphorism of Edsger W. Dijkstra about testing: “Program testing can be used to show the presence of bugs, but never their absence” [43].
- Finally, software testing has a fixed budget. The testing budget is bounded by the three corners of the Project Management Triangle: Project Cost, Project Scope, and Time to Market. In fact, testing is often poorly performed or skipped all together because of its high cost and the pressure of time to market.

Over the last three decades, a considerable amount of research has attempted to mitigate the enormous cost of inadequate software testing infrastructures and the intrinsic labour-intensiveness of the testing process [9], [70]. The primary goal has been to improve testing automation by focusing on the following two objectives:

- Develop advanced techniques to automate the test input generation process.
- Find innovative support procedures to automate the software testing process.

Nevertheless, all such efforts have had only limited impact on the software industry, where testing activities remain largely reliant on human intervention [8].

In response to this context, this thesis proposes scalable automated techniques, supported by a solid theoretical foundation of computer science, to implement the first objective of the goal of automating the entire testing process. In particular, we explore innovative techniques to improve automated test input generation approaches [29], [121] to strengthen structural coverage testing, which has long been advocated by the software industry to assess test adequacy [20], [56], [117], [132] but is still limited, even with the recent development of advanced techniques [105], [128], [138], [142]. We then enhance security vulnerability detection techniques [64], [93] by integrating static and dynamic program analysis techniques to actively search for security vulnerability defects. The techniques developed in this thesis furnish current approaches with the ability to exploit program dependencies (e.g. control and data dependencies), and utilize static analysis techniques together with effective search mechanisms to handle scalability issues, which are a key limiting factor facing current dynamic testing approaches [29], [35], [65].

1.2 Scope

Software testing is a quality control function of quality assurance which is combined with software engineering to improve the quality and reliability of software. Software testing incorporates several testing activities together with a number of testing types. A specific testing type is designed to find specific types of errors and is performed distinctly at distinct testing levels. Methodologies, techniques, tools, and the humans involved in a particular testing type can vary significantly. This thesis mainly covers the following topics in the software testing field, namely automated generation of test inputs, structural coverage testing, and security vulnerability testing.

1.2.1 Test Input Generation

Test input generation is a fundamental activity of software testing. It involves the design of test inputs to execute the piece of software under consideration. With a given test input, the execution of the software will exercise some software behaviour and provide feedback to software engineers to evaluate the software validity against desired test requirements. At the same time, the design of test inputs must attempt to achieve the highest likelihood of finding the most errors with the minimal amount of time and effort. Automation of test input generation has become an active research field in the software testing research community, especially over the last decade. A number of automated test input generation techniques have been extensively explored, developed, and applied in academia, research labs, and industry. These techniques include random testing [3], symbolic execution [112], dynamic symbolic execution [29], search-based testing [87], and the chaining approach [52]. Noticeably, recent years have the development of techniques that have been applied to test industrial-scale software systems and which have uncovered many serious subtle bugs and security vulnerability defects in the Windows [66] and Linux [93] operating systems, saving millions of dollars [15].

Regardless of the many challenges remaining, the recent achievements in the automation of test input generation motivated us to intensively study the limitations of current approaches, and to extensively explore and develop effective techniques in order to significantly improve the efficiency of software testing, particularly in the field of test input generation.

1.2.2 Structural Coverage Testing

Achieving high structural coverage criteria such as statement and branch coverage is an important goal of software testing [4], [145]. A structural coverage criterion gives quantitative measures of the degree to which the source code of the program under test is exercised during software testing. Structural coverage provides mechanisms to drive software testing, specifically to determine what test inputs need to be designed for improving the adequacy of software testing so that it completely covers the structure of the program under test. Intuitively, a high structural coverage degree obtained implies that the program is more thoroughly tested and has a lower chance of containing software defects than a program with low structural coverage. Therefore, structural

coverage offers another layer of assurance to gain high confidence about software quality and reliability. In practice, structural coverage suggests stopping rules for determining whether sufficient testing has been performed so that it can be terminated.

This thesis looks closely at structural coverage criteria such as statement and branch coverage. In particular, we will explore and develop effective techniques to perform test input generation in an automated directed mechanism to significantly improve structural coverage results for the program under test. An important implication from this study is that since an error can be encoded as a code element in the program, achieving high structural coverage indirectly strengthens error-revealing capabilities.

1.2.3 Security Vulnerability Testing

A noticeable drawback of adopting structural coverage criteria in performing software testing is that although complete structural coverage might have been achieved, the correctness of the software is not directly addressed. The interesting fact is that a set of test cases is considered to be adequate with respect to a structural coverage criterion, when it simply exercises a fragment that is astronomically small over the entire state space of the program. Not surprisingly, it has been reported that when test engineers run millions of test cases and completely achieve all structural coverage criteria, faults still persist in the final software product [38], [39].

Therefore, apart from structural coverage testing, this thesis develops augmented techniques that favour fault detection. In particular, we focus on improving the capability of uncovering security vulnerabilities defects, one of the most serious classes of security threats. We integrate static runtime verification and dynamic test generation techniques to strengthen security vulnerability detection for C programs.

1.3 Thesis Details

The proposal of this thesis is aligned to the exploratory research effort to enhance the efficiency of software testing through exploring and developing techniques to improve the attainable automation of test input generation. The proposed techniques are evaluated as to their capability to effectively perform structural coverage testing and security vulnerability testing. The following subsections discuss the statement and the contribution of this thesis and outline the organization of the remainder of the thesis.

1.3.1 Thesis Statement

Developed based on the automated test input generation technique of dynamic symbolic execution, the thesis looks closely at improving the efficiency of path exploration to effectively and efficiently achieve test objectives. The driving force is yet simple. When testing sizable and complex real world programs, the path space of the program under test can be exceedingly huge to be systematically and exhaustively explored. Whenever test objectives can be reduced to a reachability problem, which simply requires generating test inputs to execute specific code elements, search techniques can be developed to break down the path space and guide path exploration to propagate selected aspects of semantics in order to influence the executability of code elements. The search techniques to be developed incorporate ideas from symbolic analysis, constraint solving, dynamic program analysis, control and data dependence analysis, and static runtime verification. We will show that our proposed directed search methods are more effective than recently developed search techniques in both the capability to optimize the expensive cost of performing dynamic test input generation as well as the capability to significantly enhance structural coverage testing results and security vulnerability detection.

1.3.2 Thesis Contribution

The contribution of this thesis to existing knowledge can be summarized as follows:

- An extensive literature review of existing automated test input generation techniques is presented in chapters 2–4. Strengths and drawbacks of these techniques are compared and contrasted to establish the context that led to the undertaking of this thesis.
- A goal-oriented dynamic test generation approach is proposed in chapter 4, based on the latest advances in dynamic test generation and constraint solving technology. The proposed approach distinguishes itself from existing approaches by its ability to exploit control and data dependencies of the program to improve significantly the efficiency of path exploration toward exploring test goals.
- A structural coverage testing framework is proposed in chapter 5 based on the goal-oriented dynamic test generation approach presented in chapter 4. In this framework, the testing approach is reduced to the problem of finding test inputs to explore specific code elements in the program under test. Experiments on

simulated and real world test subjects proved that the proposed framework is efficient at improving structural coverage results when compared to existing approaches.

- A security vulnerability testing framework is proposed in chapter 6, again based on the goal-oriented dynamic test generation approach presented in chapter 4. In this framework, a test goal is a potential safety violation and the testing approach is to generate test inputs to discover the violation. We utilize static runtime verification to diagnose potential safety violations and dynamic test generation to perform test input generation. Experiments conducted against 23 buffer overflow vulnerabilities demonstrate the significant superiority of our testing framework over existing approaches.
- Two sets of experiments carried out in chapters 5 and 6 provide valuable observations in the context of developing techniques to perform dynamic test generation for testing objectives such as structural coverage criteria and security vulnerability detection.

1.3.3 Thesis Organization

The rest of the thesis is organized into the following six chapters:

Chapter 2: An introduction to software testing is provided in this chapter. Software testing is a very broad area and we therefore only discuss the basic activities, important roles, and motivating forces leading to the wide adoption of testing in practice. The demand for the automation of software testing is highlighted. The chapter continues with an extensive literature review of automated test input techniques, with a special focus on dynamic symbolic execution and its impact on academia, research labs, and industry. Finally, the fundamental scalability issue of dynamic symbolic execution is underlined and the main objectives of this thesis are presented.

Chapter 3: Based on the literature review performed in chapter 2, dynamic symbolic execution is chosen as an enabling technology in the development of this thesis. This chapter describes this technique with its programming model, execution model, and the interplay between concrete and symbolic execution. It also illustrates the inherently complex path-based analysis, a key limiting factor of dynamic symbolic execution. This limitation is the primary motivation that led to the proposal of this thesis.

Chapter 4: After looking closely at the execution mechanism of dynamic symbolic execution, a goal-oriented dynamic test generation approach is proposed in this chapter, in which the goal is a code element in the program and the testing approach is to generate test inputs to explore the goal. The key element underlying this approach is a search algorithm named GUIDER. It is driven by the chaining mechanism and based on dynamic symbolic execution to perform path exploration for uncovering specific test goals. GUIDER distinguishes itself from existing search algorithms in three major aspects: (1) it mitigates the path explosion problem by centralizing on data dependences which truly affect the executability of the test goal; (2) it is able to refine path exploration when the local search space is saturated; and (3) it determines control dependences on the fly and exploits the static program structure to optimize path exploration. This search algorithm is an underlying component for the two testing frameworks proposed in chapters 5 and 6.

Chapter 5: Based on the goal-oriented dynamic test generation approach presented in the chapter 4, a structural coverage testing framework, named SCT, is proposed in this chapter. SCT reduces the testing problem to a search problem, where the search task is to perform dynamic symbolic execution-based path exploration of given code elements. The performance of SCT is evaluated on simulated and real world test subjects and compared to popular search algorithms for structural coverage testing.

Chapter 6: Similarly, a security vulnerability framework, named SEBO, is proposed in chapter 6. The primary focus of SEBO is to test buffer overflow vulnerability defects in C programs. SEBO uses DEPUTY [31]—a novel type system for pointers—to identify potential runtime violations and dynamic test generation to find test inputs to uncover actual violations. SEBO has considerable ability to detect errors caused by erroneous pointer arithmetic operations on buffers. An experimental evaluation conducted against 23 buffer overflow vulnerabilities demonstrates a significant improvement of SEBO over popular approaches.

Chapter 7: The conclusion is presented in this chapter, followed by a summary of the work conducted in this thesis. Finally, ideas for future work are presented.

1.3.4 Publications

The following research papers have been written during the course of this research thesis.

1. "Goal-Oriented Dynamic Test Generation", under submission
2. TheAnh Do, A.C.M. Fong, Russel Pears, "Dynamic Symbolic Execution Guided by Data Dependency Analysis for High Structural Coverage", *L.A. Maciaszek and J. Filipe (Eds.): Communications in Computer and Information Science 410, Springer, Heidelberg*, 2013, pp. 1–13.
3. TheAnh Do, A.C.M. Fong, Russel Pears, "Precise Guidance to Dynamic Test Generation", *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering*, Wroclaw, Poland, 29–30 June 2012.
4. TheAnh Do, A.C.M Fong, Russel Pears, "Scalable Automated Test Generation Using Coverage Guidance and Random Search", *Proceedings of the 34th International Conference on Software Engineering, the 7th IEEE/ACM International Workshop on Automation of Software Test*, Zurich, Switzerland, 2–3 June, 2012.
5. TheAnh Do, A.C.M. Fong, Russel Pears, "How Effective is Model Checking in Practice?", *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering*, Beijing, China, 8–11 June 2011, pp. 239–244.

Chapter 2

Software Testing

We begin by quoting the opening of the Preface to Roger S. Pressman's *Software Engineering – A Practitioner's Approach* [108]

> When computer software succeeds—when it meets the needs of the people who use it, when it performs flawlessly over a long period of time, when it is easy to modify and even easier to use—it can and does change things for the better. But when software fails—when its users are dissatisfied, when it is error prone, when it is difficult to change and even harder to use—bad things can and do happen. We all want to build software that makes things better, avoiding the bad things that lurk in the shadow of failed efforts. To succeed, we need discipline when software is designed and built. We need an engineering approach.

An engineering approach to software is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software [75]. The foundation underlying such an approach is a quality focus that supports software engineering. In the attempt to support the development of high quality software, software testing is an indispensable element of any software engineering approach. Software testing includes activities in the software development life cycle that aim to measure and improve quality of software.

A very commonly asked question is "*Why does software fail?*" Obviously, the engineering of software is a human-originated activity where opportunities for the inclusion of human fallibilities are numerous. In order words, software is fallible because of human fallibilities. There are risks associated with software failure. Software development is therefore accompanied by a quality assurance activity to assess such risks. Testing is a critical element of software quality assurance intended to reduce risks of software failures. Unfortunately, the cost required to perform software testing is great, and can be up to half of the total software development costs. Huge efforts are therefore being directed at exploring and developing innovative techniques to enhance the efficiency of software testing and to improve the quality of software throughout the whole development process.

This chapter provides a basic overview of software testing and presents the "big picture" of which this research project is a part. We carry out a critical literature review to evaluate the existing research in software testing. From there we establish the whole context underlying the development of this thesis.

Section 2.1 describes software testing; it covers activities in software testing as well as testing levels to be performed throughout the software development life cycle. The demand for automation of software testing is also illustrated in this section. In section 2.2, the literature review focuses on automated test input generation techniques, and the capability of these techniques is compared and contrasted from practical perspectives. These techniques include random testing [3], symbolic execution [112], dynamic symbolic execution [26], [62], [124], search-based testing [87], and the chaining approach [52]. The theoretical foundation of dynamic symbolic execution and the chaining approach is separately presented in chapters 3 and 4, respectively.

Section 2.3 presents our study objectives in the development of this thesis. The first objective is to develop a goal-oriented dynamic test generation approach; the second objective focuses on structural coverage testing; and the last objective attempts to enhance buffer overflow vulnerability detection, one of the most serious classes of security threats [41]. Finally, a summary of the chapter is given in section 2.4.

2.1 Software Testing – The Whole Context

Since software is written by humans, software products have defects due to fallibilities introduced by humans. In addition, pressures such as deadlines, the complexity of systems, and technology changes all bear down on software engineers and increase the likelihood of errors, which spread throughout specification, design, and implementation phases in the software development process. These errors are the root cause from which system failures begin. Specifically, if a software specification harbouring an error is used to specify a component, then the component will be faulty. If the faulty component is implemented into a system, the system may cause failures. Unfortunately, software failures have the potential to be catastrophic.

Software failures may seriously affect businesses and customers. While businesses may encounter substantial financial consequences and suffer reputational damage, customers, in almost every case, may confront stress and inconvenience. Examples of software failures are depressingly common. A software error in the baggage handling system postponed the opening of Denver's airport for a full 16 months, at a loss of US$1.1 million per day throughout the postponement [21]. A software glitch in the trading firm Knight Capital's newly installed software system resulted in a US$440 million loss in just 45 minutes [125]. The glitch was triggered by a flawed software algorithm that bought the shares at market price then sold them at the bid price. The loss was greater than the firm's revenue in the second quarter of 2012 and threatened the stability of the firm.

On 4 June 1996, the Ariane 5 rocket crashed just 37 seconds after launch [143]. The crash was due to an integer overflow occurring when converting a 64-bit floating point into a 16-bit integer value in the control software of the rocket. Further notorious examples include the Mars Climate Orbiter disintegration [33] and the FBI Virtual Case File project abandonment [59].

More seriously, a software flaw in the control part of the radiation therapy machine Therac-25 caused the death of six cancer patients between 1985 and 1987 after they were exposed to an overdose of radiation [22].

With the increasing reliance on information processing in the functioning of software systems, the magnitude and complexity of these systems are growing rapidly. Software

systems no longer exist independently, but are typically embedded in a larger context, connecting and interacting with several other components and systems. They therefore become much more vulnerable to errors. The delivery of software systems with low defects is an enormously challenging and complex activity due to the ever-growing complexity, the limit on project budget, and the pressure to drastically reduce system development time or time to market. The reliability of software systems is a key issue in the software development process.

In attempting to develop high quality software, software engineers need to bring a systematic, disciplined, and quantifiable approach with discipline, adaptability, and agility to the design and building of software systems. Software engineering encompasses processes, methods, and tools grounded on a quality focus to enable complex computer-based systems to be developed in a timely manner. The software development process incorporates five activities — communication, planning, modelling, construction, and deployment. These define a framework for the effective delivery of software engineering technology. This framework forms the basis for management control of software projects and grounds the context in which technical methods are applied, work products are produced, milestones are established, change is properly managed, and importantly quality is ensured.

In order to ensure software quality, the software development process is accompanied by a quality assurance activity. Quality assurance establishes an infrastructure that supports solid software engineering methods, rational project management, and quality control actions, all crucial for building high quality software. Software quality assurance explicitly defines software quality, creates a set of activities to exhibit high quality, performs quality control and assurance activities on software projects, and uses metrics to develop strategies for measuring and improving the software development process and thereby the quality of software upon deployment.

In software quality assurance, testing is a quality control function with the primary goal of finding software errors. Motivating forces for software testing to enter the quality assurance picture are the costly consequences associated with software failures. The intent of software testing is therefore to find the highest possible number of errors where reviews and other software quality assurance activities are not sufficient. The following subsection provides a full overview of the software testing process.

2.1.1 Software Testing Activities

Software testing is an important component in all software engineering approaches. In early stages of software engineering, engineers attempt to design and build software from an abstract concept to a working product. In the software testing stage, engineers create a set of test cases that are intended to deconstruct the software that has been built. In fact, software testing is a constructive process with the aim of improving software quality that attacks the software product in order to reveal its flaws and weaknesses.

Since testing is a quality control function of software quality assurance that aims to reduce risks associated with software failures, the process of carrying out software testing is grounded by the following primary objectives [86]:

- Testing is the process of executing a program with the intent of finding errors.
- A good test case is one that has a high probability of detecting an as yet undiscovered error.
- A successful test case is one that detects an as yet undiscovered error.

The overriding motivation for software testing is to design test cases that rigorously and systematically uncover various classes of errors with a minimal amount of time and effort. Additionally, testing examines software functionality with respect to specifications, and evaluates behavioural and performance requirements. Consequently, the reliability and quality of software can be measured through data collected during the process of software testing.

Software testing constitutes a significant part of any software engineering project. It is not unusual for software development companies to spend between 30% and 50% of the total software project costs on testing. In safety-critical systems such as chemical plants, nuclear power plants, and flight control systems, software testing can cost three to five times as much as all other software engineering stages combined. This enormous cost stems chiefly from the fact that human intervention features in almost every testing activity.

Specifically, software testing is a dynamic technique that takes the piece of software under consideration and provides its compiled code with inputs, or test cases. Correctness is determined by forcing the software to traverse a set of execution paths or sequences of code statements representing a run of the software. Based on the

observations during test execution, the actual output is compared to the output as documented in the system specifications. Exhaustive testing of all execution paths is practically impossible; in practice only a small subset of these paths is exercised. Hence, testing can never be complete.

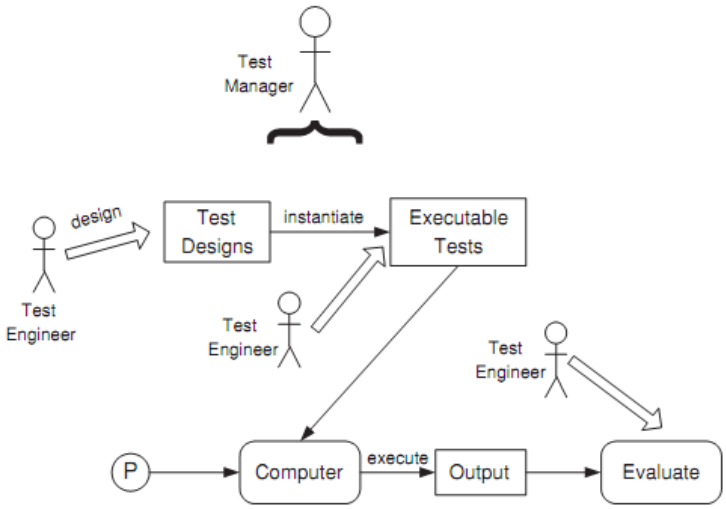

Figure 2.1: Activities of test engineers in software testing [4]

Figure 2.1 describes the three fundamental activities carried out by engineers in the software testing process: test design, test execution, and output evaluation. In test design, test engineers design test cases based on the test requirements and the source code. A test case must give a full description about testing requirements, environments, instructions and conditions, input values, and expected outputs. This description is completed with the execution details and test results from the next two activities. In test execution, test engineers execute the test cases against the software under test P and capture the execution details. In output evaluation, test engineers evaluate the outputs to determine if the test cases reveal any faults in the software.

In all three of the above activities, test engineers play a central role in driving the testing process. As software systems are growing rapidly in scale and complexity, such a human-centric process is very time consuming, expensive, and unreliable [133]. The automation of these activities is thus required to improve software quality and reliability. The next subsection provides more information about how testing is conducted throughout the software development life cycle.

2.1.2 Testing throughout the Software Life Cycle

Software testing helps all Information Technology professionals to develop higher quality software [4]. To effectively and efficiently carry out software testing, it is crucial to establish a rigorous and systematic strategy for testing software thoroughly. If testing is conducted haphazardly, time is wasted, unnecessary effort is expended, and, even worse, undetected errors remain. The strategy should provide a road map that describes the steps to be conducted in software testing, when these steps are planned and then undertaken, and how much effort, time and resources will be required [108]. A testing strategy must accommodate low level tests that are necessary to verify that a small piece of software has been correctly implemented as well as high level tests that validate major system functions against customer requirements.

In fact, a strategy for software testing is largely determined by the software development process that is applied to design and build the software. This process will define how testing is organized during software development.

Figure 2.2 illustrates a typical scenario of a software development process and how testing fits into software development activities. The software development process is depicted as a spiral. Initially, system engineering determines the role of software and leads to software requirements analysis, where the information domain, function, behaviour, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, we come to design and finally to coding. To develop computer software, we spiral further inward along streamlines that decrease the level of abstraction on each turn.

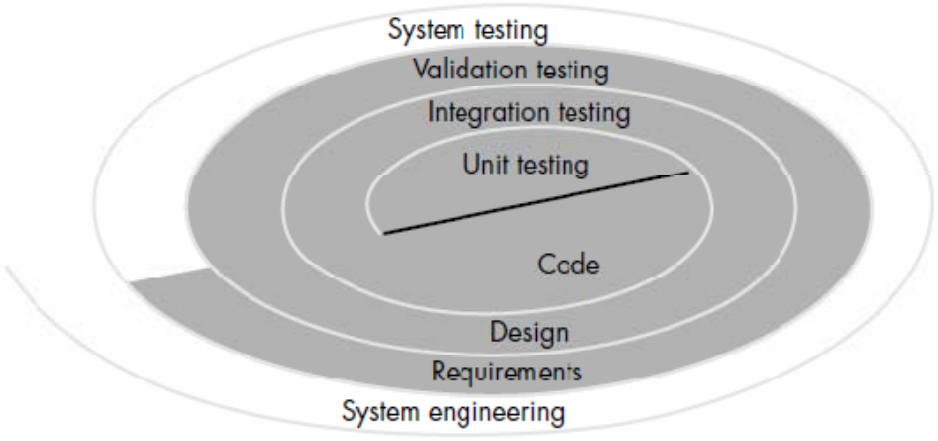

Figure 2.2: Software testing strategy [108]

Figure 2.2 also depicts the strategy for software testing in the form of a spiral. There are four test levels associated with four activities of the software development process. Each level has its own objectives and as we spiral out along streamlines the scope of testing broadens with each turn.

Unit Testing. Unit testing begins at the heart of the spiral and concentrates on each unit of the software as implemented in source code. It searches for defects and verifies the functioning of software units such as modules, programs, objects, and classes that are separately testable.

Integration Testing. Testing progresses to integration testing by moving outward along the spiral, where the focus is on design and the construction of the software architecture. Integration testing is a systematic technique for constructing the program structure while at the same time designing tests to uncover errors associated with interfacing.

Validation Testing. In validation testing, requirements established as part of software requirements analysis are validated against the software that has been constructed. Specifically, validation testing tests the software system with respect to customer needs, requirements, and business processes to determine whether or not to accept the system.

System Testing. Finally, system testing is reached by taking the final turn outward on the spiral. As software is only one element of the larger computer-based system, system testing verifies and validates the software in connecting and interacting with other system elements such as hardware, people, and databases.

Throughout the software development process, software is tested from two different perspectives: white-box testing and black-box testing. In white-box testing, test engineers know the internal workings of the software system and testing is conducted to ensure that internal operations are performed according to specifications and that all internal components have been adequately exercised. In black-box testing, the specified function that the software system has been designed to perform is known and testing is conducted to check each function is fully operational while at the same time searching for errors in each function. In both cases, the purpose is to find the maximal number of errors with the minimal amount of effort and time.

Each individual level of testing utilizes white-box and black-box testing differently. In unit testing, heavy use is made of white-box testing to exercise specific execution paths

in a component's control structure in order to improve structural coverage and error detection capability. In integration testing, since components are assembled to form the complete software package, testing addresses interfaces and interactions between components. Although white-box testing can be used to ensure coverage of major control paths, black-box testing is the most prevalent during integration. After the software has been integrated, black-box testing is used extensively during validation to validate functional, behavioural and performance requirements. Finally, system testing is conducted to verify that the software properly operates in the entire computer-based system upon deployment.

In addition, software testing is accompanied by various test types that clearly define the objective of test levels. A particular test type aims to address a specific type of errors and therefore to assess a specific software quality characteristic. A test type is associated with a test objective, for example testing of a function to be performed by the software system; of a non-functional quality factor such as reliability or usability; of the structure of the software; of software changes to confirm that defects have been fixed (confirmation testing); of unintended changes (regression testing); or of the security of the software system (security testing). Depending on the stated objectives, software testing is organized differently. By focusing software testing on a certain test type, test engineers have a clear idea of how to perform testing and what testing techniques and tools to use. This can significantly reduce the effort and time required to conduct testing. In the next subsection, we discuss the need for automation in software testing and define our research scope in the entire context of software testing.

2.1.3 Automation of Software Testing

Software failures can be very costly. Software testing is one component in the overall quality assurance activity that seeks to ensure that software systems enter service without defects that can lead to serious failures. It is one of the primary techniques adopted widely in industry to improve quality and reliability of software. As noted above, the intensive intervention of humans in software testing however makes it a laborious, unreliable, and expensive activity to carry out. In the software development process, one must balance competing demands for resources. For instance, if we need to deliver a software system faster (or in less time), it will usually cost more. In practice, software testing is bounded by the three corners of the triangle of resources, namely

time, money, and quality. These three affect one another, and also influence all the activities, including testing, to be performed in the entire process of software development. This is where automation of software testing enters the picture.

Automated software testing has gradually become as an important process that can improve the efficiency of testing in the software industry [29]. The benefit of testing automation is obvious. It relieves the testing bottleneck and achieves faster time-to-market, reduces the money spent on software testing, increases test coverage and reduces risks, configures and repeats test cases, reduces human resources, finds more and deeper defects earlier, ensures corporate compliance, ensures the scalability of system testing, and establishes consistent and thorough testing. Not surprisingly, automated testing techniques have discovered serious bugs in commonly used and well-tested software that lay hidden for years despite massive human testing efforts [66].

In the process of software testing, the design of tests can be a very challenging activity. The challenges stem chiefly from the potentially endless number of test cases from the input domain and therefore the potentially infinite number of behaviours that the software system can display. Exhaustive testing is infeasible. The test design activity must be targeted at finding the most errors with minimal amount of time and effort. In the development of the research process in this thesis, we primarily focused on generation of test inputs, the first activity of the software testing process as presented in section 2.1.1. The primary purpose has been to explore and develop effective techniques for automation of test input generation for improvements of test objectives such as structural coverage testing and security testing. The next section provides an extensive literature review of the automated test input generation techniques that have been developed over the last three decades.

2.2 Automated Test Input Generation Techniques

Given the software system under consideration, test engineers provide its compiled code with inputs, execute the software, and observe the software behaviour as well as its output to determine whether or not the software behaves properly and expectedly. Motivating forces for automation of software testing are mainly the human-intensive intervention and therefore the expensive cost of software testing. Automated techniques for generation of test inputs are one of the most important steps toward automating the

entire software testing process. More importantly, automated test input generation can significantly enhance the efficiency of software testing many magnitudes over manual testing. It can achieve higher software coverage and discover more and deeper software errors that are not easily found by humans. This section carries out a broad literature review on automated test input generation techniques in order to establish the whole context underlying the development of this thesis. The strengths and shortcomings of individual techniques are evaluated in practical perspectives in order to assess the current state of research and to draw out research questions.

2.2.1 Random Testing

Random testing is a simple but effective technique for automated test input generation [3]. In random testing, the process of generating test inputs is performed randomly, based on sampling the interface of the software program under test. The most distinguishing features of random testing are its high precision and cost-effectiveness. This is because random testing is based on dynamic analysis, using as input values actual values to truly execute the program, and the test input generation process is often independent of the complexity and size of the program.

In random testing, researchers have mainly focused on improving the capability of the technique for error detection [17], [68]. Miller, Fredriksen, and So [91] introduced *fuzz testing* to generate random ASCII character streams and used them as inputs to test Unix utilities for abnormal terminations and non-terminating behaviours. They were able to crash 24% of the utility programs tested. In subsequent work, they extended fuzz testing to generate sequences of keyboard and mouse events, and found errors in applications running in X Windows, Windows NT and Mac OS X [54], [90]. Today, fuzz testing is used routinely in the industry to detect security vulnerabilities. The work of Kropp, Koopman, and Siewiorek [82] applies random test generation to test low level system calls while Groce, Holzmann, and Joshi [61] found dozens of errors when testing a file system used in space missions at NASA.

The success of random testing relies chiefly on its ability to explore a huge number of program behaviours without being greatly affected by the size and complexity of the program under test. However, in random testing, since test inputs are generated by randomly sampling the input space, this leads to two key problems. One is that many test inputs may exercise the same program paths and hence are redundant. The other is

that the probability of generating particular test inputs to trigger buggy behaviours or to explore corner case branches may be astronomically small. In practice, random testing often results in low code coverage [104]. This shortcoming means that potential software errors can be missed if the code containing the errors is not exercised. To illustrate, consider the probability of executing the **then** branch of the conditional statement `if (x == 10) { ... }`. In random testing, this branch has only one chance out of 2^{32} to be exercised if `x` is a randomly chosen 32-bit input value.

2.2.2 Symbolic Execution

Research in symbolic execution dates back to the 1970s and the work of Boyer, Elspas, and Levitt [14], Clarke [24], Howden [71], King [78], and Ramamoorthy, Ho, and Chen [116]. In contrast to random testing, which truly executes the compiled code of the program under test, symbolic execution is a static program analysis-based technique that analyses code statically without executing it. The key idea behind symbolic execution is to use as input values *symbolic values* instead of actual data, and to represent values of program variables as *symbolic expressions*. As a result, the test program is executed symbolically; symbolic constraints are gathered and then solved by an off-the-shelf constraint solver to obtain actual input values. The following provides a brief explanation of the execution mechanism in symbolic execution.

In the simulation of the program execution, symbolic execution maintains a symbolic memory S, which maps memory addresses to symbolic expressions, and a symbolic path constraint PC, a first-order quantifier-free formula over symbolic expressions. In this way, when an expression is evaluated, it is evaluated symbolically using symbolic expressions in S, and S is updated accordingly. Similarly, when a conditional statement **if** (e) **then** S_1 **else** S_2 is executed, PC is updated according to the **then** or **else** branch taken. If the **then** branch is taken, PC is updated to $PC \wedge \sigma(e)$; otherwise, it is $PC \wedge \neg\sigma(e)$, where $\sigma(e)$ denotes the symbolic predicate obtained by evaluating e in symbolic memory. Note that unlike concrete execution, in symbolic execution, both branches can be taken, resulting in two execution paths. PC is checked for satisfiability every time it is updated by using the underlying constraint solver. If PC becomes unsatisfiable, symbolic execution terminates along the corresponding path. After the execution of a program path, the symbolic path constraint PC symbolically simulating the execution of that path is as follows:

$$PC = \sigma_1 \wedge \ldots \wedge \sigma_{i-1} \wedge \sigma_i \wedge \sigma_{i+1} \wedge \ldots \wedge \sigma_n \quad (1)$$

Then, PC is solved by the underlying constraint solver to obtain actual input values for the simulated execution path. The simulation of symbolic execution can be organized in a form of a tree referred to as a *symbolic execution tree*. Search techniques such as depth-first and breadth-first search can be employed to perform symbolic execution in a systematic manner in order to exhaustively explore all feasible paths of the program.

Symbolic execution has a variety number of applications in program testing and analysis, for example automated test input generation [25], [112], test sequence generation [130], [140], proving program properties [13], [46], [53], [111], and static detection of runtime errors [18], [30], [36], [50], [127]. A well-known symbolic execution tool is Symbolic Java PathFinder (SPF) [6], [109], which is part of the JPF project [77] (open-sourced since 2003). SPF has been applied at NASA in various projects, such as test input generation for the Orion control software where it helped uncover subtle bugs [109], fault tolerant protocols, NextGen (TSAFE) aviation software or robot executives. SPF has been extended with a symbolic string analysis at Fujitsu, where it is being used for testing web applications [51].

Symbolic execution, while offering an effectively automated mechanism to exhaustively and systematically explore all feasible paths of the program, suffers from significant limitations in practice. First, symbolic execution works under the support of constraint solvers and hence is not able to reason about complicated pieces of the code that are not supported by the constraint theory of the underlying constraint solvers. Second, when reasoning about loops or recursion, symbolic execution has to put a limit on the search depth since it is not able to detect the number of iterations of such programming structures. This results in incomplete reasoning and thus may reduce its capability in detecting potential errors and/or achieving high structural coverage. Third, symbolic execution is not able to deal with native libraries. Finally, symbolic execution is limited due to the combinatorial explosion problem of the program path space. These inherent limitations significantly limit the applicability of symbolic execution when dealing with real world software systems which are often complex and large-scale.

In the context of using symbolic execution to automate the test input generation process, symbolic execution lends itself particularly well to this task, since a symbolic path

constraint to reach a branch or statement in the code, when solved, provides exactly the test input to exercise the statement or branch. In addition, symbolic execution is used in regression testing, generating test inputs to expose the changes in evolved software [1], [107]. All these applications of symbolic execution to real world software, however, are bounded because of its inherent limitations.

2.2.3 Dynamic Symbolic Execution

Dynamic symbolic execution is receiving a considerable amount of attention in current industrial practice [19], [28], [30], [62], [64], [76], [93], [95], [109], [119], [120], [124], [128]. It intertwines the strengths of random testing and symbolic execution to achieve the scalability and high precision of dynamic analysis, and the power of the underlying constraint solver. In dynamic symbolic execution, the program under test is executed simultaneously both concretely and symbolically in a *directed automated* mechanism. One of the most important insights of dynamic symbolic execution is the ability to reduce the execution into a mix of concrete and symbolic execution when facing complicated pieces of code, which are the critical obstacle to *classical* symbolic execution.

The key idea underlying the development of dynamic symbolic execution is to perform concrete execution simultaneously with symbolic execution [62]. Specifically, the program under test is initially executed on a given or randomly generated test input, and symbolic predicates are collected to form a symbolic path constraint PC along the being executed path. After an execution, one can pick up a symbolic predicate, negate it, and then form a constraint system to be solved by the underlying constraint solver. The satisfiability of the constraint system results in a new test input. The next execution of the test program with this input will follow the previously executed path up to the corresponding conditional statement of the negated symbolic predicate, but afterward will change the flow of control to the other branch. As a result, for every symbolic path constraint, the number of program paths to be executed can be 2^n, or be exponential in the number of symbolic predicates. In practice, the number of symbolic predicates in the program can be extremely large (often infinite), especially in the presence of loops and/or recursions, causing dynamic symbolic execution to encounter combinatorial explosion of the path space. This is the fundamental scalability issue of dynamic symbolic execution-based approaches [29], [35], [65].

Dynamic symbolic execution, since proposed by Godefroid, Klarlund, and Sen in 2005 [62], has inspired the development of new research directions and many techniques, tools, and applications [29], [121]. It has been applied to test many industrial software systems and has detected serious bugs. In the following, we provide a brief review of a few dynamic symbolic execution-based tools, and their impact in academia, research labs, and industry.

CUTE and jCUTE. CUTE (A Concolic Unit Testing Engine for C) [124] extends Godefroid *et al*.'s Directed Automated Random Testing (DART) [62] to handle input recursive data structures and represent several optimizations for constraint solving. In CUTE, pointer constraints are separated from numeric (integer) constraints, and are represented and solved approximately. jCUTE (CUTE for Java) [120] combines dynamic symbolic execution with dynamic partial order reduction to systematically generate both test inputs and thread schedules for multi-threaded programs. Both tools have been applied to test several open-source software, including *java.util* library of *Sun JDK 1.4* and have detected several previously undocumented bugs. CUTE and jCUTE have also been used to study dynamic symbolic execution in different courses at several universities.

CREST. CREST [19] is an automated test input generation tool for C, based on dynamic symbolic execution. It is an extensible platform for building and experimenting with search heuristics for achieving high code coverage. Since being released as open–source software in May 2008 [23], CREST has been used by several research groups. For instance, CREST has been used to build tools for regression testing [139], for detecting SQL injection vulnerabilities [118], and for identifying infeasible code to achieve high structural coverage [12], and has been modified to run distributed on a cluster for testing a flash storage platform [81]. In the development of this research project, we also used CREST for structural coverage testing and security testing.

SAGE. SAGE [64] is an *automated white-box fuzz testing* tool for finding security vulnerabilities in software. It incorporates fuzz testing with recent advances in symbolic execution and dynamic test generation to scale to large-file parsers embedded in applications with millions of lines of code and execution traces with billions of machine instructions, such as Microsoft Excel. SAGE was developed at Microsoft Research and has detected many new security vulnerabilities in hundreds of applications. Notably,

SAGE found roughly one third of all the known bugs discovered by file fuzzing during the development of Microsoft's Windows 7 [15], [66], saving millions of dollars by avoiding expensive security patches for nearly a billion PCs worldwide. The approach has since been adopted in several tools and has been applied to test Linux applications [93], including codecs, image reviewers and media players.

PEX. PEX [128] is a dynamic symbolic execution-based tool for automated test input generation for .NET code, supporting languages such as C#, VisualBasic and F#. PEX has implemented a rich set of basic search strategies and offers a fair choice among them in order to avoid any bias toward particular control flows for maximal code coverage. PEX introduces *Parameterized Unit Testing* [73] to allow software unit testing in two aspects: the specification of the behaviour of the system and the test inputs to cover a particular implementation. PEX is a Visual Studio 2010 Power Tool which is used inside Microsoft. PEX is also available for academic purposes.

Dynamic symbolic execution has been shown to be an effective technique for automation of test input generation. When applied to practice, however, the technique has a number of limitations. Firstly, dynamic symbolic execution is limited because of the difficulty of programming language features. Complex constraints, data structures, and native calls of real world software systems all result in imperfect symbolic execution. It is very difficult to capture the semantics of object-oriented programming features such as abstraction, inheritance, polymorphism, and encapsulation with symbolic modelling. Even though dynamic symbolic execution is language-independent, it is usually implemented in low level programming languages such as C, x86 assembly, Java bytecode, and Common Intermediate Code in .NET.

Secondly, like symbolic execution, dynamic symbolic execution is bounded by the theory supported by the underlying constraint solver. For example, the technique cannot precisely deal with arithmetic operations with floating numbers since most constraint solvers do not support floating point theory. Thirdly, the unavailability of source code limits dynamic symbolic execution from fully executing symbolic execution in order to construct complete path constraints on individual program executions. Consequently, dynamic symbolic execution is only able to explore part of the entire symbolic execution tree of the test program.

With these three limitations, dynamic symbolic execution reduces to a partial form of symbolic execution and random testing. The power of the underlying constraint solver is not fully exploited, which compromises the technique's success. In the most extreme cases, dynamic symbolic execution reduces to a complete form of random testing. Its final limitation is the fundamental scalability problem. In fact, the combinatorial explosion of the path space is one of the biggest challenges facing dynamic symbolic execution. In practice, the technique simply exercises a limited number of program paths. This phenomenon will limit the efficiency of software testing in both structural coverage testing and error detection capabilities. Exploring and developing effective techniques to improve the efficiency of path exploration over the program path space is critical to applying dynamic symbolic execution in practice, and this was one of the primary motivations for the proposal of this research project.

2.2.4 Search-Based Testing

Search-based testing [87] formulates the process of generating test inputs as a search problem [27], [69]. It uses fitness functions to measure the improvement of the search process, and meta-heuristic search techniques such as Hill Climbing, Simulated Annealing, and Evolutionary Algorithms to find test inputs. The search space is the space of possible inputs of the program under test. Most research on search-based testing focuses on automating the test input generation process in software testing [87].

The most distinguishing feature of search-based testing is, unlike symbolic execution and dynamic symbolic execution which depend mostly on the capability of constraint solvers to generate test inputs, the process of finding test inputs itself. Therefore, search-based testing can handle a wide range of primitive input types such as integral and floating-point numbers. Furthermore, the use of fitness functions provides suitable measurement for targeting particular code elements of testing. These features of search-based testing have been exploited to improve dynamic symbolic execution toward achieving high structural coverage by guiding path exploration [141] and by solving floating point computations [84].

The applicability of search-based testing in automation of test input generation is limited in several aspects, however. Firstly, the process of finding test inputs involves using meta-heuristic search strategies, which are performed largely randomly. Secondly, if search-based testing is applied to test complex and sizeable software where the cost of

running programs is expensive, the process of finding test inputs implemented in this technique may require considerable resources to explore structural coverage elements. Finally, the use of fitness functions suffers significantly from the flag problem [16], where fitness functions cause a flat fitness landscape, giving no guidance to the search process. Flags, however, appear very often in real world software [16].

2.2.5 The Chaining Approach

The chaining approach proposed by Ferguson and Korel [52] is another approach for automation of test input generation. This technique relies on a local search method called the *alternating-variable method* to find test inputs for maximal code coverage. A noticeable feature of the chaining approach is that it introduces a chaining mechanism for exploring code elements in a systematic manner. Specifically, to explore a given code element, the chaining mechanism identifies statements leading up to the target structure, which may influence the outcome of the code element. Those statements are sequences of events that the search process must proceed along to target the code element.

The chaining mechanism can therefore be considered as a slicing technique [131] which simplifies programs by focusing on selected aspects of semantics. As a result, the chaining approach can provide precise guidance for the search process since it forces the consideration of data flow analysis, and it is effective since it slices away irrelevant code segments to the execution of the code element. These two strengths can guide the search process into potentially unexplored but promising areas of the path space to unclose hard-to-reach code elements.

The chaining mechanism is more suitable for goal-oriented test input generation approaches. In practice, however, this approach faces two main limitations. One is that the chaining mechanism makes use of data dependency analysis to guide the search process; the data dependency analysis, however, is done statically and therefore may not be precise. The other limitation is that the process of finding test inputs using the alternating-variable method is performed largely randomly.

2.3 Objectives of the Study

Automation of software testing is an important requirement for improving the quality and reliability of software in industry. It mitigates the hardship of manual testing, which is labour-intensive and error-prone, and alleviates the expense of testing, which often accounts for around 50% of the total software development costs. Testing automation can be enhanced by automating the process of generating test inputs. Over the last three decades, techniques have been proposed to achieve this goal, ranging from random testing [3], symbolic execution [112], dynamic symbolic execution [26], [62], [124], search-based testing [87], and the chaining approach [52].

Among these proposed techniques, dynamic symbolic execution has been demonstrated to be an effective technique for automation of test input generation. The technique has been applied to test many industrial software systems and uncover "million-dollar" bugs [66]. However, the fundamental scalability issue limiting the capability of dynamic symbolic execution is its combinatorial explosion of the path space, which is extremely huge (often infinite) in sizeable and complex software. This phenomenon has been significantly highlighted in several research studies:

> … path explosion represents one of the biggest challenges facing symbolic execution, and given a fixed time budget, it is critical to explore the most relevant paths first. [35]

> A significant scalability challenge for symbolic execution is how to handle the exponential number of paths in the code. [29]

> In theory, systematic dynamic test generation can lead to full program path coverage, i.e., program verification. In practice, however, the search is typically incomplete both because the number of execution paths in the program under test is huge … [65]

The impact of this particular limitation of dynamic symbolic execution on the efficiency of software testing is significant. If dynamic symbolic execution is carried out in a way that exhaustively and systematically explores all feasible paths of the program under test, then it often ends up with only small regions of the code explored. Consequently, in practice the objective of achieving high structural coverage of software testing is hard to realize using dynamic symbolic execution. More importantly, the capability of detecting errors can be limited since the unexercised code may harbour errors. An

important observation in relation to this is that structural coverage testing, such as statement or branch coverage, requires each code element (e.g. statement or branch) to be executed *only once*, i.e. dynamic symbolic execution can be performed so as to cover every selected code element rather than attempting to explore the entire path space of the program under test. This may lead to a significant reduction in the number of program paths that need to be explored.

In order to improve the applicability of dynamic symbolic execution in industrial software development practice, the development of effective techniques to strengthen the efficiency of path exploration over the entire program path space has become one of the most important driving forces in the research community [29], and was actually the primary motivation behind the proposal of this research project. Specifically, the development of this research project is aligned to the following three main objectives.

The first objective of this study is to develop a goal-oriented dynamic test generation approach, where a goal is a specific code element and the testing approach is to find test inputs to exercise the goal. This is referred to as the reachability problem in computer science. In general, reachability is an undecidable problem, but it has a number of important applications in program analysis. At its core, our approach proposes a search algorithm that significantly improves the efficiency of path exploration toward effectively and efficiently exploring test goals. Specifically, it exploits program dependencies (e.g. control and data dependencies) to precisely identify the root cause leading to the execution of test goals and then effectively guide path exploration to uncover them with minimal exploration effort. This is an attempt to address the path explosion limitation facing dynamic symbolic execution. A detailed explanation of our approach is presented in chapter 4.

The second objective of this study is to develop a framework for structural coverage testing. Structural coverage testing is an important element of software testing in software engineering approaches. As exhaustive testing is potentially endless, structural coverage provides stopping rules for determining whether sufficient testing has been carried out and whether it can be terminated. Structural coverage testing gives measurements of test quality where a degree of adequacy associated with a test set can give a level of confidence about the correctness of the software under test. A high coverage degree implies that the program is more thoroughly tested and has a lower

chance of containing software defects than a program with low structural coverage. In this framework, the goal-oriented approach proposed above is used to perform dynamic symbolic execution-based path exploration for effectively improving structural coverage results. A detailed description of this framework is given in chapter 5.

The third and final objective of this study is to develop a security vulnerability testing framework. With the development of the Internet, the application of software systems has suffered seriously from security problems. Security vulnerabilities are reported every day in commonly used software [37], [101]. Such vulnerabilities can be exploited by attackers to demolish data and the functionality of computer systems, causing significant financial losses and potentially endangering lives. The development of effective techniques to identify and eliminate security bugs is vital to protect the software product before it can be deployed for use on the Internet. In this framework, we propose combining static runtime verification and dynamic symbolic execution to improve security vulnerability detection capabilities. The development of the framework is centred on the use of the goal-oriented approach proposed above to significantly strengthen the ability of dynamic symbolic execution for quickly uncovering security defects such as buffer overflow vulnerabilities. A full description of the framework is given in chapter 6.

2.4 Summary

Software is designed and constructed by humans where opportunities for the inclusion of errors are numerous. An error in software leads to a defect, which can cause an observed failure. Software failures may have catastrophic consequences such as financial losses and even cause the loss of lives. Software testing is a quality control function of quality assurance, indispensable to software engineering approaches. The primary objective of software testing is to find the maximal number of errors with a minimal amount of time and effort. Testing is a human-intensive activity, however, and manual testing is laborious, unreliable, and expensive to perform. Therefore, automation of software testing has been developed as one of the key means of improving the quality and reliability of software.

The development of this research project focused on the process of test input generation, which lies at the heart of the software testing process. Specifically, this research project

studies the automated test input generation technique of dynamic symbolic execution, and explores and develops effective techniques to perform dynamic symbolic execution so that the efficiency of software testing is maximized. The research project will aim to improve the degree of the attainable testing automation for effectively and efficiently performing structural coverage testing as well as security vulnerability testing.

This chapter began by providing a basic understanding of software testing, covering testing activities and testing levels throughout the software development life cycle (section 2.1). The pressing need for the automation of software testing was also discussed. Section 2.2 presented an extensive literature review of the automated test input generation techniques developed over the last three decades. It compared and contrasted the various techniques as to their strengths and shortcomings. Through this critical review, we solidly established the context underlying the development of this research project, as described in section 2.3, and formulated three research objectives.

The first objective is to develop a goal-oriented dynamic test generation approach for effectively and efficiently finding test inputs to cover specific elements in the program's source code. The second objective is to develop a framework for structural coverage testing. The final objective is to strengthen security vulnerability detection capabilities. These three objectives are satisfied in chapters 4, 5, and 6, respectively. The core technique forming the development of this thesis is dynamic symbolic execution. The next chapter therefore describes the theoretical concepts behind dynamic symbolic execution to establish the background for this study.

Chapter 3

Dynamic Symbolic Execution

This chapter describes dynamic symbolic execution, a powerful program analysis technique introduced by in work of Godefroid, Klarlund, and Sen [62], and of Cadar, Ganesh, Pawlowski, Dill, and Engler [30] in 2005.

The power of dynamic symbolic execution lies in the novel intertwinement of dynamic and static program analyses, the increased availability of computational power, and the constraint solving technology. Dynamic symbolic execution is now the underlying technique of several popular testing tools [29]. It has been applied to challenging application domains and has uncovered subtle bugs in commonly used software [66].

The chapter is divided into the following sections. Section 3.1 gives an overview of dynamic symbolic execution. Section 3.2 introduces an imperative programming language on which the execution model of dynamic symbolic execution is formally established. In section 3.3, we describe the four important components in dynamic symbolic execution, these being concrete execution, symbolic execution, test input generation, and search algorithm. This is followed by a formalization of depth-first search in section 3.4 to illustrate the execution model of dynamic symbolic execution

and demonstrate its improvements over classical symbolic execution. The fundamental scalability limitation of dynamic symbolic execution is presented in section 3.5 and a summary of the chapter is given in section 3.6.

3.1 Overview

Dynamic symbolic execution is adopted in this research to offer a practical trade-off between dynamic analysis and static analysis [62]. In essence, the novelty of the technique is the back-and-forth interaction between random testing and symbolic execution, meaning that the weaknesses of one technique can be mitigated greatly by the strengths of the other, and vice versa. This is obvious when observing the two techniques with respect to the following points: precision and test input generation capability, as shown in Table 3.1.

Table 3.1: A summary of strengths and weaknesses of random testing and symbolic execution

	Precision	**Test Input Generation**
Random Testing	High	Not effective Redundant
Symbolic Execution	Low	Effective Not redundant

Therefore, in the execution of dynamic symbolic execution, the purpose is to intertwine the high precision of dynamic analysis and the capability of test input generation of static analysis, and this is precisely reflected in the following execution model. An execution in dynamic symbolic execution involves executing the program under test with concrete values while capturing the semantics of executed instructions along the execution using symbolic values. So, while the former drives the execution, the latter is to model the execution in forms of a function of symbolic input values. The result is a *path condition*, a conjunction of symbolic predicates, determining the executed program path. By negating one predicate and solving the corresponding path constraint with an off-the-shelf constraint solver, one can obtain a new test input to steer the execution along an alternative path. This procedure is often performed under the guidance of a search algorithm to explore the path space of the program under test in a desired exploration order.

3.2 Programming Model

To formalize dynamic symbolic execution, its execution model and algorithms, we introduce an imperative programming language. The programming model of the proposed language is simple yet expressive enough to cover the basic operations of real programming languages in order to convey important notations when illustrating dynamic symbolic execution.

For simplicity, we limit primitive data types to integral numbers only. A program P in this programming language is represented as a tuple (V, V_0, L, l_0, ξ, E) where:

- V is a set of variables,
- $V_0 \subseteq V$ is a set of input variables,
- L is a set of control locations,
- $l_0 \in L$ is the start location,
- ξ is a labelling function which labels each location $l \in L$ with one of the following basic operations: (1) a termination operation **halt**, (2) an assignment operation $v := e$, where $v \in V$ and e is an arithmetic expression free of side effects over V, and (3) a conditional operation **if** (e) **then** l' **else** l'', where e is a Boolean expression over V, and l' and l'' are locations in L,
- $E \subseteq L \times L$ is a set of edges such that (1) every location l where $\xi(l)$ is an assignment operation has only one location l' with $(l, l') \in E$, and every location l where $\xi(l)$ is a conditional operation **if** (e) **then** l' **else** l'' has two edges (l, l') and (l, l'') in E. For an assignment operation $\xi(l)$ of location $l \in L$, we use $\eta(l)$ to denote l's unique sink location.

In this representation of the program P, control locations correspond to program instructions while edges correspond to control flows indicating the execution flow from one instruction to another. For simplicity, we assume that there is only one location $l_{\mathbf{halt}}$ in program P where $\xi(l_{halt})$ = **halt**. An execution of program P on input I proceeds through a sequence of labelled control locations $l_1, l_2, l_3, \ldots, l_n$ where $l_1 = l_0$ is the start location and $l_n = l_{\mathbf{halt}}$ is the termination location. This location sequence is referred to as an executed program path, or simply a path. For convention, the two control locations of a condition operation **if** (e) **then** l' **else** l'' are called branches in which l' represents the *true* branch and l'' represents the *false* branch.

3.3 Execution Model

Given a program under test P, in dynamic symbolic execution, P is executed simultaneously both concretely and symbolically. Concrete execution carries out the execution on concrete input values and defines a uniquely executed program path. Symbolic execution captures the semantics of every executed operation along the executed path using symbolic input values. This side-by-side execution model is sketched in the ExecuteProgram algorithm in Figure 3.1.

Algorithm 1 ExecuteProgram

Input : Program P = (V, V_0, L, l_0, ξ, E), Input *input*
Output : Path condition φ

```
1:  for each v ∈ V do
2:      M(v) := input(v)
3:      if v ∈ V0 then S(v) := α_v end if
4:  end for
5:  φ := true ; l := l0
6:  while ξ(l) ≠ halt do
7:      switch ξ(l) do
8:          case v := e :
9:              M := M[ v ↦ M(e) ]
10:             S := S[ v ↦ S(e) ]
11:             l := η(l)
12:         end case
13:         case if (e) then l′ else l″ :
14:             if M(e) then
15:                 φ := φ ∧ S(e) ; l := l′
16:             else
17:                 φ := φ ∧ ¬S(e) ; l := l″
18:             end if
19:         end case
20:     end switch
21: end while
22: return φ
```

Figure 3.1: Execution model of dynamic symbolic execution

The algorithm takes as input a program P and a test input *input*, and outputs a path condition φ. A detailed explanation of the algorithm is provided in the following sections where we proceed to formalize the four basic components composing dynamic symbolic execution, starting from concrete execution, symbolic execution, test input generation, to search algorithm. And finally, a summary on the interaction between concrete execution and symbolic execution is presented.

3.3.1 Concrete Execution

The semantics for the concrete execution of the program is captured using a *concrete memory map* M, which is a mapping of variables in V to actual values. We denote a memory mapping using $M' \coloneqq M[\ m \mapsto v\ ]$, where M' is the same memory map as M, except that $M'(m) = v$. For an expression e, we use M(e) to denote the value obtained by evaluating e in M where every variable v appearing in e is substituted by M(v).

As shown in the ExecuteProgram algorithm, the execution model for concrete execution proceeds in the following way. Initially, it creates a concrete memory map M_0 where input variables in V_0 are set to initial actual values in *input* and variables in $V \setminus V_0$ to default constant values. The control location is set to the start location $l = l_0$. As the execution goes forward, each operation is executed, and the concrete memory map and the control location are updated correspondingly. Suppose the current memory map is M and the current control location is l, then under the programming language introduced, we consider the following situations:

- If $\xi(l)$ is an assignment operation $v \coloneqq e$, l is updated to $\eta(l)$ and M is updated to $M[\ v \mapsto M(e)\ ]$.
- If $\xi(l)$ is a conditional operation **if** (e) **then** l' **else** l'', there are two possibilities that the execution can branch. If M(e) is evaluated to **true**, then l is updated to l'; otherwise, M(e) is **false** and l is updated to l''. In the two cases, the concrete memory map M is not changed.
- If $\xi(l)$ is the termination operation **halt**, the execution terminates.

Naturally, the execution model of the program under concrete execution traverses a unique sequence of labelled control locations, e.g. $l_1, l_2, l_3, \ldots, l_n$, or a path. The path is the only one instance over the many program paths and is defined only if the program input is given. In fact, a concrete execution is not intended to capture or synthesize any semantic information of executed operations. However, it is fully *automated* as long as the program under consideration is compiled.

To explore the path space of the program, a test input generation mechanism is needed to carry out concrete execution. Obviously, the mechanism should take advantage of the currently executed path to improve path exploration. In the following section, we describe how symbolic execution can enable such a test input generation mechanism.

3.3.2 Symbolic Execution

The semantics for the symbolic execution of the program is captured using a *symbolic memory map* S and a *path condition* φ. The symbolic memory S is a mapping from variables in V to symbolic expressions over symbolic input values. Operations on S are basically similar to those on concrete memory M. The path condition φ is a conjunction of symbolic predicates over symbolic input values, where symbolic predicates are collected along an execution.

The execution model for symbolic execution proceeds in the following way. Initially, it creates a symbolic memory map S where input variables in V_0 are set to initial symbolic values and variables in $V \setminus V_0$ to default constant values. The path condition φ is set to **true** and the control location is set to the start location $l = l_0$. As the execution goes forward, each operation is executed, and the symbolic memory map and the control location are updated correspondingly. Suppose the current memory map is S and the current control location is l, then under the programming language introduced, we consider the following situations:

- If $\xi(l)$ is an assignment operation $v := e$, l is updated to $\eta(l)$ and S is updated to $S[\, v \mapsto S(e) \,]$.
- If $\xi(l)$ is a conditional operation **if** (e) **then** l' **else** l'', there are two possibilities that the execution can branch. If M(e) is evaluated to **true**, then l is updated to l' and φ is updated to $\varphi \wedge S(e)$; otherwise, M(e) is **false**, and l is updated to l'' and φ is updated to $\varphi \wedge \neg S(e)$. In the two cases, the symbolic memory map S is not changed.
- If $\xi(l)$ is the termination operation **halt**, the execution terminates.

Obviously, the execution model of the program under symbolic execution is basically to simulate a concrete execution using symbolic input values. This is obvious when considering the execution of a condition operation **if** (e) **then** l' **else** l''. Here, the value of e is evaluated in the *concrete* memory mapping and is used to decide the execution flow, whether jumping to l' or l''.

The result of symbolic execution is a path condition φ; it is constructed by taking a conjunction of symbolic predicates whenever a conditional operation is executed. The path condition synthesizes the executed path and will be used for test input generation.

3.3.3 Test Input Generation

An execution of the program in dynamic symbolic execution results in a path condition with the following form:

$$\varphi = \sigma_1 \wedge ... \wedge \sigma_{i-1} \wedge \sigma_i \wedge \sigma_{i+1} \wedge ... \wedge \sigma_n \quad (1)$$

It is a conjunction of symbolic predicates σ_i ($1 \leq i \leq n$) and characterizes the executed program path. An important property of the path condition is that every single symbolic predicate of φ can provide an opportunity to execute the program along an alternative path. Therefore, the test input generation procedure in dynamic symbolic execution is performed based upon this property. Specifically, this procedure performs the following mechanism. Given a path condition φ, it selects a symbolic predicate, e.g. σ_i, negates it, and forms the constraint system $\varphi' = (\sigma_1 \wedge ... \wedge \sigma_{i-1} \wedge \neg\sigma_i)$. Then, the procedure interacts with an underlying constraint solver to check the satisfiability of φ'. If φ' is satisfiable, then a solution, an input, is returned back to the procedure. The most important characterization of the input obtained by this mechanism is that the execution of the program with this input will follow the previous path up to the corresponding conditional operation of the negated predicate σ_i, but afterward change the flow of control to execute the alternative control location. This mechanism is often performed under the guidance of search strategies to direct dynamic symbolic execution for exploring the path space in the desired exploration order.

The interaction of the underlying constraint solver in test input generation is crucial. In fact, the effectiveness and efficiency of dynamic symbolic execution is based heavily on the following aspects of the constraint solver:

- The theory supported—a powerful theory supported by the constraint solver will enable dynamic symbolic execution to deal with a wide range of application domains with high complexity.
- The constraint solving capability—given a constraint system to be solved, the capability of the constraint solver to quickly find a satisfying solution is a key factor in performing dynamic symbolic execution.

A perfect constraint solver does not exist in practice. The presence of concrete execution is critical in dealing with the high complexity of real world software.

3.3.4 Generic Search Algorithm

So far, we have seen that in dynamic symbolic execution the program under test is executed concretely and symbolically. The result of one execution is a path condition characterizing the executed program path. Based on the path condition, the test input generation procedure is invoked to select a symbolic predicate and form a constraint system to be solved by the underlying constraint solver for test inputs. However, we have not yet discussed in what way a particular symbolic predicate is to be selected to continue dynamic symbolic execution. Or more precisely, how is the path space of the program under test to be explored?

We now introduce a generic search algorithm to perform path exploration, which is shown in Figure 3.2. This algorithm can be instantiated to explore the program path space in any desired exploration manner.

Algorithm 2 GenericSearch

Input : Program P = (V, V_0, L, l_0, ξ, E), Path condition φ
Output : Set of inputs T

```
1: while termination conditions are not reached do
2:     σ_i := SelectPredicate(φ)
3:     if there exists input t satisfying constraint (σ_1 ∧ ... ∧ σ_{i-1} ∧ ¬σ_i) then
4:         T := T ∪ { t }
5:         φ' := ExecuteProgram(P, t)
6:         ProceedSearch(P, φ')
7:     end if
8: end while
9: return T
```

Figure 3.2: Generic search algorithm to perform dynamic symbolic execution

The input to the GenericSearch algorithm is the program under test P and the current path condition φ; however, it is parameterized by the following three components:

- A termination criterion determining when to terminate the search (line 1).
- A selection process determining how a specific symbolic predicate is selected to continue the search (line 2).
- A procedure determining how the newly constructed path condition φ' is to be processed to continue the search (line 6).

In practice, we typically activate a search algorithm on an input with either randomly generated input values or zeros. The search terminates when either the termination criterion is reached or the path space of the program is exhaustively explored. The output can be a set of test inputs generated during the search together with information captured during dynamic symbolic execution, e.g. program crashes, assertion violations, and non-termination.

The design of the SelectPredicate procedure is undeniably the key to the successful implementation of dynamic symbolic execution. It determines the effectiveness of the technique in coping with the large size and high complexity of real world software. In the next section, we introduce depth-first search and use it to illustrate the execution model of dynamic symbolic execution.

3.4 Depth-First Search

In the implementation of dynamic symbolic execution, depth-first search (DFS) is a standard search algorithm used to carry out path exploration. DFS has been implemented in testing tools to explore the path space of the program under test and has been shown to be effective through exposing previously unknown serious software defects in commonly used software. The algorithm skeleton is given in Figure 3.3; it is similar to the GenericSearch algorithm given in the previous section.

Algorithm 3 DepthFirstSearch

Input : Program P = (V, V_0, L, l_0, ξ, E), Path condition φ,
Last negated predicate *last*
Output : Set of inputs T

```
1:  index := Length(φ)
2:  while index ≥ last do
3:      σ_i := SelectPredicate(φ, index)
4:      if there exists input t satisfying constraint (σ_1 ∧ ... ∧ σ_{i-1} ∧ ¬σ_i) then
5:          T := T ∪ { t }
6:          φ′ := ExecuteProgram(P, t)
7:          DepthFirstSearch(P, φ′, index + 1)
8:      end if
9:      index := index – 1
10: end while
11: return T
```

Figure 3.3: Depth-first search for performing dynamic symbolic execution

DFS explores the path space of the program in a depth-first order in which the *last* and *not-yet-negated* symbolic predicate is always chosen to be negated. Theoretically, DFS offers a systematic search mechanism to explore all feasible program paths. The input to the DepthFirstSearch algorithm is the program under test P, the current path condition φ, and the last predicate *last*. *last* is actually an index indicating the lower bound of the counter *index* to iterate over the path condition φ for exploring the path space, i.e. $last \leq index \leq \text{Length}(\varphi)$. A call triggering DFS has the form DepthFirstSearch(P, φ_0, 1), where P is the program under test, φ_0 is the initial path condition obtained by executing P concretely and symbolically on a randomly generated test input, for example, and the last predicate is set to 1.

DFS explores the path space starting from the last predicate of φ to *last* (lines 1–2). For each predicate σ_i chosen in this depth-first order, σ_i is negated to form a constraint system to be solved for a test input t (lines 3–4). With this test input t, P is executed to obtain another path condition φ'. Then, DFS recursively calls itself with (*index* + 1) to explore the path space further generated by the newly executed path φ'. This procedure is repeated until the whole path space of the program is exhaustively explored or the termination criterion is reached. For simplicity, the latter case is not explicitly specified in the algorithm.

DFS is straightforward to implement and has been adopted widely to illustrate the execution model of dynamic symbolic execution. In the following sections, we employ DFS to illustrate dynamic symbolic execution through three examples to demonstrate its significant improvements over classical symbolic execution.

3.4.1 Example 1

```
void example01 (int x, int y) {
    int z;
    z = 100;
    if (x == z + z) {
        if (y == z * z) {
            ERROR();
        }
    }
    return;
}
```

Figure 3.4: Example illustrating the execution model of dynamic symbolic execution

Consider the C program `example01` in Figure 3.4, which takes as inputs two integer variables `x` and `y`. The input parameter vector is $\overrightarrow{M}_0$ = <x, y>. To start dynamic symbolic execution, let us assume that the randomly generated concrete value for `x` is 111222 and that of `y` is 222111, i.e., $\vec{I}$ = <111222, 222111>. Then, the concrete memory mapping is M = [x ↦ 111222, y ↦ 222111]. And the symbolic memory mapping is S = [x ↦ X_0, y ↦ Y_0] where X_0 and Y_0 are initial symbolic values for `x` and `y`. With this configuration, the program execution executes the **else** control location of the outer **if** operation and terminates at the **halt** operation. We obtain the path condition φ = <¬(X_0 = 200)>, and memory mappings M = [x ↦ 111222, y ↦ 222111, z ↦ 100] and S = [x ↦ X_0, y ↦ Y_0, z ↦ 100]. An attempt to negate the only last symbolic predicate of the path condition results in a constraint system <(X_0 = 200)>, which forces the underlying constraint solver to return the solution $\vec{I}'$ = <X_0 = 200>. Then, we update the input vector $\vec{I}$ by replacing the value of every variable v appearing in $\vec{I}$ with S(v) where S(v) is evaluated based on both the current input vector $\vec{I}$ and the solution $\vec{I}'$. In this case, we obtain `x` = 200 and `y` = 222111, that is, $\vec{I}$ = <200, 222111>.

With the attained input vector, we form the configuration M = [x ↦ 200, y ↦ 222111] and S = [x ↦ X_0, y ↦ Y_0] to continue dynamic symbolic execution. The program execution with this configuration executes the **then** control location of the outer **if** operation instead and then the **else** control location of the inner **if** operation. We obtain φ = <(X_0 = 200) ∧ ¬(Y_0 = 10000)>, M = [x ↦ 200, y ↦ 222111, z ↦ 100], and S = [x ↦ X_0, y ↦ Y_0, z ↦ 100]. By negating the last symbolic predicate and forming the constraint system <(X_0 = 200) ∧ (Y_0 = 10000)>, we obtain the solution $\vec{I}'$ = <X_0 = 200, Y_0 = 10000>. This leads to the input vector $\vec{I}$ = <200, 10000>.

Finally, the program execution with the configuration constructed based on the input vector $\vec{I}$ = <200, 10000> results in the path condition φ = <(X_0 = 200) ∧ (Y_0 = 10000)>. The execution executes the **then** control location of the two **if** operations and hits the **halt** operation `ERROR()`. At this point, the program path space has been exhaustively explored and DFS terminates.

This example shows that, like symbolic execution, the notation of path condition plays a central role in driving dynamic symbolic execution to explore the program path space.

3.4.2 Example 2

```
void example02 (int x, int y) {
    int z;
    z = x*x*x + 1;
    if (y == z) {
        ERROR();
    }
    return;
}
```

Figure 3.5: Example illustrating how dynamic symbolic execution deals with complex computational expressions

Consider the C program `example02` in Figure 3.5, which takes as inputs two integer variables `x` and `y`. The input parameter vector is $\vec{M}_0$ = <x, y>. Let us assume that the initial program input to start dynamic symbolic execution is $\vec{I}$ = <100, 200> where x = 100 and y = 200. The configuration to execute the program is M = [x ↦ 100, y ↦ 200] and S = [x ↦ X_0, y ↦ Y_0]. Now, as the execution goes forward, each operation is executed concretely and symbolically. For the assignment operation `z = x*x*x + 1`, in concrete execution, it adds the mapping M(z) = 1000001 to the concrete memory map M, that is, M = [x ↦ 100, y ↦ 200, z ↦ 1000001]. In symbolic execution, it evaluates the right-hand side of the assignment and forms the mapping S(z) = X_0^3 + 1. Now, we suppose that the underlying constraint solver does only support the *linear arithmetic theory*. That is, the computation of the cube expression X_0^3 + 1 goes beyond the theory that can be handled by the underlying constraint solver. We consider below how classical symbolic execution and dynamic symbolic execution behave to deal with this situation.

In classical symbolic execution, whenever the expression to be evaluated is outside the scope of the theory supported by the underlying constraint solver, the technique skips executing not only the currently executed operation but also any further operations followed from this operation. That is, only part of the code is considered in classical symbolic execution. In this particular example, the presence of the cube expression leads to bypassing the execution of the assignment operation and consequently bypassing the whole program.

In dynamic symbolic execution, if an expression in symbolic execution cannot be modelled by the theory of the underlying constraint solver, then the concrete memory map M is used to simplify the expression so that it can be reasoned by the constraint solver. This simplification feature works to symbolically and concretely evaluate the cube expression `x*x*x + 1`. The expression has left-to-right associativity and is evaluated in the following order:

$$(((x *^1 x) *^2 x) +^3 1)$$

where the number associated with each operator determines the operator's precedence to be evaluated. For the multiplication operator $*^1$, its left-hand operand in the symbolic memory map has the value $x = S(x) = X_0$, and, similarly, its right-hand operand has the value $x = S(x) = X_0$. The evaluation of $(x *^1 x)$ yields X_0^2, X_0^2 is not a linear arithmetic expression, however. The simplification involves one of the operator's operands, e.g. the right-hand operand, being replaced with the value in the concrete memory map instead. As a result, the right-hand operand has the value $x = M(x) = 100$, and $(x *^1 x) = X_0 * 100$.

The multiplication operator $*^2$ evaluates the expression $((X_0 * 100) *^2 x)$. Like $*^1$, here the simplification replaces the right-hand operation of $*^2$ with its value in the concrete memory map, i.e. $x = M(x) = 100$ and $(((x *^1 x) *^2 x) = ((X_0 * 100) *^2 x) = X_0 * 100 * 100$. And finally, for the addition operator $+^3$, the result is $(((x *^1 x) *^2 x) +^3 1) = X_0 * 100 * 100 + 1$. The symbolic memory map S becomes $S = [x \mapsto X_0, y \mapsto Y_0, z \mapsto 10000 * X_0 + 1]$.

The execution goes forward to execute the **else** control location of the **if** operation and terminates at the **halt** operation. We obtain $\varphi = \langle \neg(Y_0 = 10000 * X_0 + 1) \rangle$. An attempt to negate the only last symbolic predicate of the path condition forms the constraint system $\langle (Y_0 = 10000 * X_0 + 1) \rangle$ to be solved by the underlying constraint solver. One possible solution could be $\vec{I}' = \langle X_0 = 0, Y_0 = 1 \rangle$. The update to the input vector yields $\vec{I} = \langle 0, 1 \rangle$, or $x = 0$ and $y = 1$. Finally, the program execution with the configuration constructed based on the obtained input executes the **then** control location of the **if** operation and hits the **halt** operation `ERROR()`. Here, DFS terminates since the path space of the program has been exhaustively explored.

This example illustrates one of the most important aspects of dynamic symbolic execution. Concrete execution simplifies symbolic execution whenever the evaluation of expression in symbolic execution goes beyond the theory that can be reasoned by the underlying constraint solver. This feature is a significant improvement of dynamic symbolic execution over symbolic execution for dealing with great complexity of real world software applications.

3.4.3 Example 3

```
void example03 (int x, int y) {
    int z;
    z = hash(x) + 1;
    if (y == z) {
        ERROR();
    }
    return;
}
```

Figure 3.6: Example illustrating how dynamic symbolic execution deals with function calls without the availability of source code

Consider the C program `example03` in Figure 3.6, which is similar to the `example02` program in the previous example except that the execution of the assignment operation in this example involves calling the function `hash()`. Here, the source code of the function is not available but its binary is provided to perform the linking process when compiling the program. The presence of the function `hash()` is an instance of a library function—vendors' code, or code that is not provided because of security reasons. This appears very often in programming practices, where developers make use of many function calls through knowing their API (Application Programming Interface) but not their source code.

Classical symbolic execution gets stuck executing the assignment operation as symbolic execution is not able to reason the `hash()` function without touching its source code. Consequently, it bypasses this operation as well as the code followed from this operation, or completely ignores the whole program.

In dynamic symbolic execution, however, since the function's binary is available, concrete execution is able to execute the function call `hash(x)` and to form a mapping for variable `z` in the concrete memory map. Now, like the previous example, in

symbolic execution, the concrete value of `hash(x)` is used to evaluate the right-hand side of the assignment and form a mapping for variable `z` in the symbolic memory map. This allows dynamic symbolic execution to execute the assignment operation and finally completely explore the path space of the program.

This example illustrates the capability of dynamic symbolic execution in dealing with the unavailability of source code. In the context of real world software applications, where library functions are often used extensively, this capability allows dynamic symbolic execution to improve the precision of the analysis as compared to symbolic execution alone.

3.4.5 Interplay of Concrete and Symbolic Execution

In summary, dynamic symbolic execution intertwines the strengths of random testing and symbolic execution to achieve the scalability and high precision of dynamic analysis, and the power of constraint solving technology. The intertwinement is expressed in the interaction between concrete execution and symbolic execution during side-by-side concrete and symbolic evaluation. On the one hand, the presence of concrete execution is essential in simplifying symbolic evaluations when encountering high complexity code. On the other hand, the presence of symbolic execution allows test input generation to be performed in an automated directed mechanism to explore the path space of the program in a desired exploration order, which is far powerful than the randomness nature of traditional random testing.

We now summarize the interplay between concrete execution and symbolic execution with the following points:

- Given a test input, concrete execution executes the program under test by traversing only one path over the program path space.
- With the executed path in concrete execution, symbolic execution executes each program operation symbolically to form a path condition. The path condition is a conjunction of symbolic predicates, characterizing the executed program path and allowing path exploration to be performed in a directed manner. That is, test inputs are generated by systematically exploring program paths at the symbolic level and these test inputs are guaranteed to traverse along the pre-determined paths.

- Concrete execution simplifies symbolic execution whenever symbolic execution goes beyond the theory that can be handled by the underlying constraint solver. In this case, the concrete values of variables in the concrete memory map are used to evaluate symbolic expressions. This results in a partial form of symbolic execution, allowing dynamic symbolic execution to deal with high complexity code in real world software applications.
- Based on constructed path condition in symbolic execution, dynamic symbolic execution interacts with the underlying constraint solver under the guidance of search algorithms to obtain new test inputs. These test inputs trigger executing many different program paths over the path space.

3.5 The Path Space Explosion Problem

Dynamic symbolic execution displays remarkable improvements over existing automated test input generation techniques such as random testing and symbolic execution. The technique does, however, reveal considerable limitations when being applied to real world software applications. Most challengingly, the fundamental scalability limitation of dynamic symbolic execution is how to handle the combinatorial explosion of the path space, which is extremely large or infinite in sizeable and complex programs. This issue has been repeatedly highlighted in the literature, as noted in the previous chapter [29], [35], [65].

From the theoretical perspective, this limitation is easy to understand. The result of an execution in dynamic symbolic execution is a path condition, a conjunction of symbolic predicates $\varphi = \sigma_1 \wedge ... \wedge \sigma_{i-1} \wedge \sigma_i \wedge \sigma_{i+1} \wedge ... \wedge \sigma_n$. Every single symbolic predicate of φ represents one possibility to execute the program along an alternative path (section 3.3.3). That is, for every path condition of length n, the number of program paths can be 2^n or be *exponential* in the number of symbolic predicates. Moreover, the negation of a symbolic predicate, e.g. σ_i, may yield another path condition of the form $\varphi' = \sigma_1 \wedge ... \wedge \sigma_{i-1} \wedge \neg\sigma_i \wedge \sigma'_{i+1} \wedge ... \wedge \sigma'_m$ where the newly collected symbolic predicate conjunction $\sigma'_{i+1} \wedge ... \wedge \sigma'_m$ may give rise to another 2^{m-i} program paths, as does any of the predicates σ'_{i+1}, …, σ'_m of the path condition φ'. That is, there is a *combinatorial* exponential growth in the number of program paths to exhaustively explore all feasible

paths of the program. This is referred to as the path space explosion problem in dynamic symbolic execution.

From the practical perspective, this limitation arises chiefly because of the presence of loops and recursion in the program under test. These programming features, however, are used extensively to implement algorithms for conveying software requirements in practice. Unfortunately, with the current computational power, even a single small fragment of code can yield a number of program paths that is too huge to be explored exhaustively. The `example04` function in Figure 3.7 illustrates this phenomenon well.

```
typedef enum {false, true} bool;
#define N 20
bool example04 (int A[N]) {
    bool success = true;
    for (i = 0; i < N; i ++) {
        if (A[i] != 25)
            success = false;
    }
    if (success) {
        // target
    }
    return success;
}
```

Figure 3.7: Example illustrating the combinatorial explosion of the path space in dynamic symbolic execution

The function takes as input an array of 20 elements and checks if all elements equal 25. This yields 2^{20} (= 1,048,576) paths with just 20 symbolic predicates. In practice, this problem becomes worse because the input of the program can be a stream of data with a too large (or unknown) size. In cases of *check_ISBN* and *check_ISSN* in the set of test subjects in our first evaluation on structural program coverage (section 5.4), for instance, both functions take as input an array with (4093 + 3) tokens, which gives rise to approximately $2^{(4093+3)}$ paths, making dynamic symbolic execution ill-suited for the goal of exploring all feasible paths of the program within the limited resources available, e.g. CPU, memory and time.

This scalability limitation in dynamic symbolic execution has been extensively researched in three main directions — (1) path space reduction, (2) parallel symbolic execution, and (3) path exploration prioritization. In the first direction, researchers aim

to prune the path space that needs to be explored [5], [58]. This can be accomplished by encoding the already explored symbolic states and preventing them from being re-explored later. In the second direction, researchers seek to exploit the increased availability of computational power to enlarge the path exploration process [126]. A theoretically desired property these two research directions have in common is striving for *completeness* of analysis. In practice, however, the symbolic execution-based path-based analysis is typically incomplete. This stems chiefly from the following reasons:

- The path space is exploded exponentially.
- Symbolic execution and constraint solving are imprecise.
- Testing has a time limit.

In this context, research in the last direction thus looks for techniques to *de facto* maximize the confidence in the testing process instead. In particular, most proposed techniques within a given testing limit attempt to prioritize path exploration in order to improve structural program coverage and enhance error detection capability.

However, the most challenging task in developing a path exploration prioritization strategy is how to mine appropriate paths to guide dynamic symbolic execution. A key objective of this research project is therefore to develop algorithms to perform path exploration toward effectively and efficiently executing dynamic symbolic execution. To achieve this, we focused on the following two attributes in the testing process: structural coverage improvements and error revealing improvements.

3.6 Summary

This chapter has covered the automated test input generation technique of dynamic symbolic execution in depth. It started by illustrating the motivation of intertwining random testing and symbolic execution in section 3.1. In section 3.2, an imperative programming language was introduced to formalize the execution model for dynamic symbolic execution. Based on this programming language, a formal representation of the four basic components composing dynamic symbolic execution was provided in section 3.3, these being concrete execution, symbolic execution, test input generation, and search algorithm. In section 3.4, depth-first search, a standard search algorithm in performing dynamic symbolic execution, was described. This was followed by three examples demonstrating improvements of the technique over classical symbolic

execution. The interaction of concrete execution and symbolic execution is the most important characteristic of dynamic symbolic execution. It is the key that allows the technique to deal with high complexity code in real world software applications in a way that goes beyond the capability of traditional symbolic execution approaches. In section 3.5, the fundamental scalability limitation, the biggest challenge facing dynamic symbolic execution, was detailed. The practical impact of this limitation on the applicability of dynamic symbolic execution is tremendous. Microsoft Research, for example, has 100+ machines running the SAGE system performing dynamic symbolic execution [66]. Coping with this fundamental scalability problem therefore emerged as a primary objective during the course of this research project.

Chapter 4

Goal-Oriented Dynamic Test Generation

In the previous chapter, we introduced dynamic symbolic execution, a powerful program analysis technique to automate the test input generation process of software testing. However, as chapter 3 identified, one of the biggest challenges facing dynamic symbolic execution is the combinatorial explosion of the path space. Much research therefore has extensively investigated techniques to improve the efficiency of path exploration in performing dynamic symbolic execution [29], [35], [66]. This is also the primary objective of this research project.

In this chapter, the intertwinement of the chaining approach [52], [92] with dynamic symbolic execution is proposed to effectively and efficiently perform the test input generation process in the context of goal-oriented testing. A goal-oriented test input generation approach has a number of practical applications in software development. The employment of the chaining approach is to direct path exploration toward better exploring a code element (or a test goal) given in the program. One of the most important features of the chaining approach is the ability to identify data dependences affecting the execution of the test goal and to carry these data dependences up to the goal structure to influence the test goal. Based on the chaining approach and the

directed search feature of path exploration in dynamic symbolic execution, we develop a goal-oriented dynamic test generation approach to significantly strengthen the efficiency of path exploration for exploring test goals. The effectiveness of the proposed approach will be assessed through coverage improvements and security vulnerability detection capability in chapters 5 and 6, respectively.

This chapter is structured as follows. Section 4.1 introduces basic concepts to formally present algorithms and formalizations presented in this chapter. Section 4.2 explains the motivation for the development of a goal-oriented test input generation approach. We also present the rationale for employing the chaining approach to carry out the dynamic symbolic execution-based path exploration process for better exploring a given test goal. Section 4.3 describes in depth theoretical aspects of the chaining approach, specifically the search mechanism and the event sequence generation process. An extended chaining approach is then presented in section 4.4 to further improve the event sequence generation process by taking into account transitive data dependences. In section 4.5, a goal-oriented dynamic test generation approach is formally presented. The approach exploits both control and data dependences to optimize path exploration in dynamic symbolic execution. And finally, a summary of the chapter is provided in section 4.6.

4.1 Background

A program structure is represented by a graph model. A *control flow graph* (CFG) of a program is a directed graph G = (N, E, s, e) where:

- N is a set of nodes where each node n ∈ N corresponds to a statement in the program, e.g. an assignment statement, an input or output statement, or the predicate of a conditional or loop statement, in which case it is referred to as a *branching node*. In the C program `example01` in Figure 4.1, nodes 2, 3, and 5 are branching nodes.
- E ⊆ N × N is a set of edges where each edge $e = (n_i, n_j) \in E$ corresponds to a possible transfer of control from node n_i to n_j. An edge $e = (n_i, n_j)$ is referred to as a *branch* if its source node n_i is a branching node. The branch executed when the condition at the branching node is true is referred to as the *true branch*. Conversely, the branch executed when the condition is false is referred to as the *false branch*. The predicate determining whether a branch is taken is referred to

as a *branch predicate*. The branch predicate of the true branch (2, 3) in the program in Figure 4.1 is `(r1 == 0)`. The false branch predicate is `(r1 != 0)`.

- s ∈ N and e ∈ N are unique entry and unique exit nodes, respectively.

```
Node  typedef enum {false, true} bool;
(s)   void example01 (int r1, int r2) {
(1)       bool invalid = false;

(2)       if (r1 == 0) {
(3)           if (r2 == 0) {
(4)               invalid = true;
              }
          }

(5)       if (invalid) {
(6)           ERROR();
          }

(7)       return;
(e)   }
```

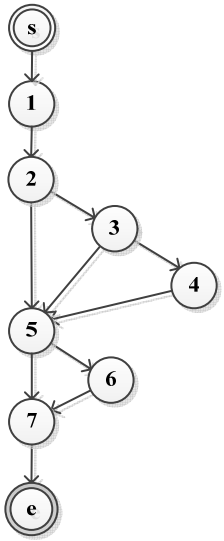

Figure 4.1: A C program and its control flow graph to illustrate basic concepts and notations in goal-oriented dynamic test generation

An *input vector* I is a vector I = (x_1, x_2, …, x_n) of input variables of the program. The domain D_{x_i} of input variable x_i, $1 \leq i \leq n$, is the set of all values which x_i can hold. The domain of the program is the cross product $D = D_{x_1} \times D_{x_2} \times \ldots \times D_{x_n}$, where each D_{x_i} is the domain of the input variable x_i. A *program input* **x** is a single point in the *n*-dimensional input space D, $\mathbf{x} \in D$.

A *path* through a CFG is a sequence P = <n_1, n_2, …, n_k> such that for every i, $1 \leq i < k$, (n_i, n_{i+1}) ∈ E. P = <1, 2, 3, 4, 5, 6, 7> is a path in the CFG of the program in Figure 4.1. A path is *feasible* if there exists a program input on which the path is traversed during the program execution; otherwise, the path is *infeasible*.

A *definition* of a variable v is a node which assigns a value to variable v. In particular, a definition of variable v can be: (1) an assignment statement, or (2) an input statement. A *use* of a variable v is a node in which v is referenced. In particular, a use of can be: (1) an assignment statement, (2) an output statement, or (3) the predicate of a conditional or loop statement. In the program in Figure 4.1, nodes 1 and 4 are definitions of variable `invalid`, and node 5 is a use of `invalid`.

A *definition-clear path with respect to a variable v* is a path in which v is not modified along the path. For example, in the program in Figure 4.1, the path <2, 5> is definition-clear with respect to the variable `invalid`, but <2, 3, 4, 5> is not, since `invalid` is defined at node 4. Similarly, a *definition-clear path with respect to a set of variables S* is a path in which none of the variables from S are modified along the path.

Control dependence captures the dependence between branching nodes and nodes being chosen to be executed by these branching nodes. The control dependency definition is given in the work of Ferrante, Ottenstein, and Warren [55]. Let X, Y, and Z be three nodes and (X, Y) be a branch of X. Node Z *postdominates* node X if and only if Z is on every path from X to the exit node e. Node Z postdominates branch (X, Y) if and only if Z is on every path from X going through branch (X, Y) to the exit node e. Z is *control dependent* on X if and only if Z postdominates one of the branches of X and Z does not postdominates X. In the program of Figure 4.1, node 6 is control dependent on node 5 because node 6 postdominates branch (5, 6) and node 6 does not postdominate node 5.

4.2 Motivation

Throughout the course of this research project, our main objective was to develop techniques to effectively and efficiently perform the test input generation process for automating software testing. The research problem we have aimed to solve is in the context of *goal-oriented testing*. Stated formally:

— Given a test goal g (e.g. statement or branch) in the program P, the goal is to find a test input t on which g is executed.

This is known as the *reachability problem* and is an *undecidable problem* in computer theory. Since a test goal can be guarded by conditions that express specific properties of the current program state, this reachability problem is similar to finding a *feasible path* to trigger the program execution to enter particular program states. It has a wide range of applications in several aspects of the software development life cycle, including debugging, software testing, software measurement, software comprehension, and software maintenance.

An application scenario of the reachability problem is to assess test adequacy in software testing. Structural coverage is a useful testing metric adopted to ensure that

every single statement in the program is executed at least once. This task can be reduced to finding test inputs to execute a specific statement in the code. Another application scenario is in using static analysis-based bug finding tools. One needs to triage the bug reports, i.e. determine if the bugs correspond to actual errors or not. This task often involves finding a test input to witness the bugs reported. Finally, in software comprehension, it is often useful for uncovering under what conditions code is executed to understand complex code bases.

In the development of a goal-oriented approach of test input generation, there are three main questions to be asked, the answers to which determine the effectiveness of the approach:

- What is a test goal?
- What is a test input generation technique?
- How to perform the test input generation technique to find a test input in order to trigger the execution of the goal?

A test goal is simply a statement or a branch in the code where its complexity depends on the conditions under which a test input can be found to execute the goal. A goal can be *unreachable*, i.e. there is no feasible path traversing the goal. In this case, a formal proof is required to verify the unreachability of the goal. This is beyond the scope of this thesis, however, and the approach carried out in this research project is that if no test input can be found to execute the test goal within a given testing time limit, the approach terminates and reports failure. The overall purpose is to improve the solving of the reachability problem, but the scenario can canaries where the test goal is actually *reachable* but the goal-oriented approach being implemented is not able to trigger its execution.

Another factor to be considered is the test input generation technique. Among the techniques surveyed in chapter 2, dynamic symbolic execution has been shown to be an effective approach to automate test input generation. A noticeable feature of this technique is that it is *directed*, i.e. one can manipulate symbolic predicates of the path condition to direct path exploration in a desired order. Therefore, it is obvious in the goal-oriented approach that the dynamic symbolic execution-based path exploration process must be guided toward exploring the test goal. Consider the `example01` program in Figure 4.1, for example. Let us assume that the currently executed path is

<1, 2, 3, 5, 7> and the corresponding path condition is $\varphi = <(R_1 == 0) \wedge \neg(R_2 == 0)>$, where R_1 and R_2 are the two symbolic values of r_1 and r_2, respectively, and the test goal is node 4. Intuitively, at the branching node 3, the change of control flow from branch (3, 5) to (3, 4) can trigger the execution of the test goal. This can be easily achieved by negating the corresponding predicate, $\neg(R_2 == 0)$, of branch (3, 5) and solving the constraint system $<(R_1 == 0) \wedge (R_2 == 0)>$ with the underlying constraint solver. This results in an input $\vec{I} = <0, 0>$, where $r_1 = 0$ and $r_2 = 0$, leading to the execution of node 4.

The last factor influencing the effectiveness of a goal-oriented approach is the guiding procedure to carry out the test input generation technique toward finding a test input to trigger the execution of the test goal. In dynamic symbolic execution, the path space of the program is explored for a particular path that traverses the test goal. A major challenge arising from path exploration in dynamic symbolic execution is the combinatorial explosion of the path space as explained in the previous chapter.

When developing techniques to effectively and efficiently conduct path exploration, the execution of test goals may require very specific guidance. Consider the `example02`

```
Node  typedef enum {false, true} bool;
      #define N 20
(s)   bool example02 (int A[N]) {
(1)       bool success = true;

(2)       int i = 0;
(3)       while (i < N) {
(4)           if (A[i] != 25) {
(5)               success = false;
              }
(6)           i ++;
          }

(7)       if (success) {
(8)           // target
          }

(9)       return success;
(e)   }
```

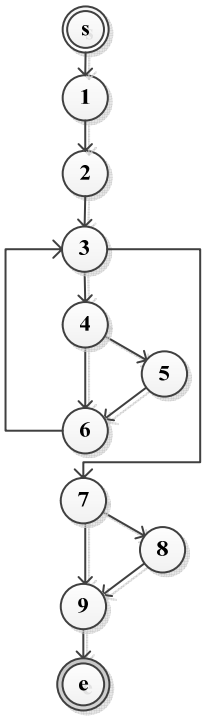

Figure 4.2: Example to illustrate difficulties of dynamic symbolic execution-based path exploration in goal-oriented approach

program in Figure 4.2, in which the test goal is to explore branch (7, 8), for example. Basically, this function is similar to the `example04` function in Figure 3.7, which takes as input an array of 20 elements and yields 2^{20} (= 1,048,576) paths with just 20 symbolic predicates. Now consider the executability of branch (7, 8). The first observation is that this branch does not constitute any symbolic predicate because its conditional expression depends on the locally declared variable `success`; any attempt to change its alternative branch to trigger its execution will fail. The second observation is that among the 1,048,576 paths, there is only one path that executes all false branches at the conditional statement of node 4 to propagate the desired `true` value of `success` down from node 1 to node 7 to execute branch (7, 8). These observations demonstrate difficulties in developing path exploration algorithms where the execution of code does not depend directly on the symbolic input. This is widely adopted in programming practices, however. For instance, Cadar *et al.* [30], when testing a number of medium-sized applications, found that less than 42% of the executed statements depended on the symbolic input. Independently, Binkley *et al.* [16] studied the testability transformation problem in search-based testing and observed that the use of Boolean-typed variables complicated test input generation and degraded program testability. Of the 23 buffer overflow vulnerabilities in our second evaluation on security vulnerability detection in chapter 6, none depends directly on the symbolic input.

To cope with such challenges, we now introduce the chaining approach [52] in an attempt to establish a search mechanism to direct dynamic symbolic execution toward effectively and efficiently exploring test goals. Given a test goal to explore, the chaining approach first performs data dependence analysis to identify statements that affect the execution of the test goal, and then uses these statements to create sequences of events that are to be executed prior to the execution of the test goal. The advantage of doing this is three-fold:

- It precisely focuses on the cause of getting the test goal to be executed.
- It forms a search mechanism to effectively perform the path exploration process.
- It slices away code segments that are irrelevant to the execution of the test goal.

These three strengths together enable a search mechanism to guide the path exploration process into potentially unexplored but promising areas of the program path space to explore high complexity code.

Based on the chaining approach, a search algorithm is proposed that is guided by the chaining mechanism and is based on dynamic symbolic execution to conduct the test input generation process, in order to establish the goal-oriented testing approach for this research project. The application of the proposed approach is focused on the following two aspects of software testing: structural coverage testing and security vulnerability detection.

In fact, dynamic symbolic execution can effectively automate test input generation of software testing in a wide range of real world software applications. What is crucial however is the path space explosion in relation to the testing time limit. The development of search techniques to optimize path exploration therefore is critical when applying dynamic symbolic execution. The presence of locally declared variables, however, makes the path exploration process tremendously difficult since dynamic symbolic execution is not able to directly manipulate predicates not relating to the program input. Operations involving these local variables encode program properties that need to be validated before being able to continue the path exploration process. In the following section, we describe the chaining approach and illustrate how this approach helps to address this problem.

4.3 The Chaining Approach

The chaining approach proposed in the work of Ferguson and Korel [52] in 1996 is an alternative test input generation technique. It makes use of data dependency information to guide the search process. The basic idea is to identify statements leading up to the goal structure which may influence the outcome of the test goal. These statements are sequences of events that the search process should proceed along to trigger the execution of the test goal. An event sequence can be thought of as an abstract path. An event simply refers to the execution of a program node. By directing the search process to traverse event sequences, it can potentially propagate data dependences necessitated to uncover the test goal. The chaining approach hence can be considered as a program slicing technique [131] which simplifies the program by focusing on selected aspects of semantics. What distinguishes the chaining approach from program slicing is the way it projects the execution of a test goal. Slicing takes into account both data and control dependences, which often yields a slice too large to explore. The chaining approach focuses only on data dependences and addresses control dependences on the fly.

In general, the chaining approach starts by executing the program under consideration on an arbitrary test input. It relies on a local search method called the *alternative variable method* to find test inputs. During a program execution, the execution of each branch (n_i, n_j) is monitored by a search process to determine whether the execution should continue through this branch or whether an alternative branch should be taken. The latter circumstance can happen, for instance, when the current branch does not lead to the test goal. If an undesirable execution flow at the current branch (n_i, n_j) is observed, the alternative variable method modifies the input vector until a new test input can be found to change the execution flow at this branch. Or else, the search declares failure.

A key characteristic of the chaining approach is that failure in finding a test input to alter the execution flow at branch (n_i, n_j) will activate the consideration of data dependencies. Specifically, the chaining approach performs data flow analysis to identify statements that have to be executed prior to the execution of node n_i. These statements are used to create event sequences that direct the search process target to the test goal. Event sequences encode data dependencies that need to be propagated down to the goal structure. This noticeable feature of the chaining approach provides an attractive search mechanism to break down the large path space of the program and discover a particular path leading to the execution of the test goal.

4.3.1 Background

In this subsection, we cover basic concepts that are used to formally present the chaining approach.

An *event sequence* E is a sequence of events, $<e_1, e_2, \ldots, e_n>$, where each *event* is a tuple $e_i = (n_i, C_i)$, where n_i is a program node and C_i is a set of variables referred to as a *constraint set*. For every two adjacent events in an event sequence, $e_i = (n_i, C_i)$ and $e_{i+1} = (n_{i+1}, C_{i+1})$, there must exist a definition-clear path with respect to C_i from n_i to n_{i+1}.

The concept of event sequences is central to the chaining approach. An event sequence basically specifies an *ordered* sequence of program nodes to direct the search process. Associated with each event $e_i = (n_i, C_i)$ is a constraint set C_i that specifies the constraints imposed on the execution from the current node n_i to node n_{i+1} of the next event e_{i+1} in the event sequence. Specifically, it ensures that all variables in the constraint set C_i must not be modified during program execution between node n_i and node n_{i+1}. Such an

execution allows the effect of definition statements to be transferred up to the target structure. The following event sequence E = <(s, Ø), (1, {`success`}), (7, Ø), (8, Ø)> is an event sequence referring to the `example02` program in Figure 4.2. It consists of four events: e_1 = (s, Ø), e_2 = (1, {`success`}), e_3 = (7, Ø), and e_4 = (8, Ø). The execution order required by this event sequence is that the start node s is first executed, followed by the execution of node 1, followed by the execution of node 7, and finally the execution of node 8. During the execution of nodes 1 and 7, a constraint is imposed by requiring that the value of `success` is not modified. For the execution between nodes s and 1, and between nodes 7 and 8, there is no constraint imposed, however. A graphical representation of the event sequence E = <(s, Ø), (1, {`success`}), (7, Ø), (8, Ø)> is given in Figure 4.3.

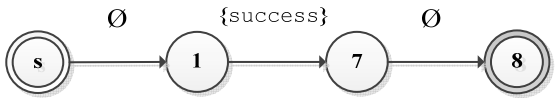

Figure 4.3: A graphical representation of event sequence E = <(s, Ø), (1, {`success`}), (7, Ø), (8, Ø)>

An event sequence is *feasible* if there exists a test input on which the event sequence is successfully traversed; otherwise, it is said to be *infeasible*. The event sequence given above <(s, Ø), (1, {`success`}), (7, Ø), (8, Ø)> is feasible; however, the event sequence <(s, Ø), (5, {`success`}), (7, Ø), (8, Ø)> is infeasible.

A *problem* node refers to a branching node for which the search process cannot find test inputs to execute an intended branch from this node.

A *last definition* n_i is a program node that assigns a value to a variable v, and this value may potentially be used by a node n_j. For the node to qualify as a last definition, a definition-clear path must exist between node n_i and node n_j with respect to v. For example, node 1 is a last definition of variable `success` at node 7 in the `example02` program of Figure 4.2. A definition-clear path with respect to `success` exists from node 1 to node 7 via the false branches of the branching node 4 through the **while** loop. A *set of all last definitions* of node n is a set of all last definitions of all variables used in n. For example, the set of all last definitions of node 7 is {1, 5}.

4.3.2 The Search Mechanism

We now describe the search mechanism in the chaining approach by using the program `example02` in Figure 4.2. The test goal is to explore node 8. For this, the chaining approach first generates the following initial event sequence:

$$E_0 = <(s, \emptyset), (8, \emptyset)>$$

This event sequence consists of the start node s and the test goal, node 8. The constraint sets associated with these events are empty. Now suppose the search process fails to find an input array with all elements equal 25 to execute the test goal, thus failing to move from node 7 to node 8. Note that the execution of node 8 can be triggered by altering the execution flow at branching node 7 from branch (7, 9) to branch (7, 8). However, no guidance is available to help the search process do so because branching node 7 involves the locally declared variable `success`. Consequently, the search process declares failure and node 7 is hence identified to be a problem node. This node is inserted before node 8 into the event sequence:

$$E'_0 = <(s, \emptyset), (7, \emptyset), (8, \emptyset)>$$

The chaining approach now performs data dependence analysis in respect of the problem node to identify last definitions that define data for variables used in the conditional expression. In this case, the conditional expression consists of only variable `success`, which is defined at nodes 1 and 5. Two event sequences are constructed accordingly, E_1 and E_2, based on the temporary event sequence E'_0.

$$E_1 = <(s, \emptyset), (1, \{\texttt{success}\}), (7, \emptyset), (8, \emptyset)>$$
$$E_2 = <(s, \emptyset), (5, \{\texttt{success}\}), (7, \emptyset), (8, \emptyset)>$$

An inserted event is formed using a last definition node and its associated constraint set is formed from the variable defined at the node. The reason for adding the last definition variable into the constraint set is to guide the search process to not modify this variable again until the problem node is encountered. By doing so, the effect of the last definition can be brought up to the problem node and hence influence the outcome of its execution flow.

Obviously, sequence E_2 cannot help to explore the test goal as the value of `success` variable is `false`, which leads to the execution of the *else* branch instead. Sequence E_1, on the other hand, guides the search process to first reach node 1 from the function

entry, which sets the value of `success` variable to the desired `true` value to explore branch (7, 8), and then continues from node 1 to node 7. When moving to node 7, the value of `success` variable may be killed at node 5 if branch (4, 5) is executed. If so, the search process is guided to change the flow of control at node 4 to execute the *else* branch, which prevents `success` variable from being set to the unwanted `false` value. This guidance is continuously refined throughout the **while** loop to preserve the constraint set {`success`} of event (1,{`success`}) while reaching to event (7, Ø). By doing so, the value of all elements in the input array is altered to 25, providing the desired input to expose the test goal, node 8.

Next, we generalize the process of generating event sequences during the search process. Given a test goal g to explore, the chaining approach begins with an initial event sequence E_0, which contains only the start node s and the test goal g, or $E_0 = <(s, Ø), (g, Ø)>$. Suppose that the search process fails to find test input to execute the event sequence due to the presence of some branching node p_1 that diverges the execution flow down an unintended branch rather than the branch that can target the test goal. Node p_1 is declared as a problem node and is inserted into event sequence E_0 to form a temporary event sequence $E_0' = <(s, Ø), (p_1, Ø), (g, Ø)>$. Then, the chaining approach performs data dependence analysis to find the set of all last definitions *lastdef*(p_1) with respect to all variables used at node p_1. For each last definition $d_i \in$ *lastdef*(p_1), a new event sequence is generated containing an event associated with that last definition:

$$E_1 = <(s, Ø), (d_1, \{ \textit{def}(d_1) \}), (p_1, Ø), (g, Ø)>$$
$$E_2 = <(s, Ø), (d_2, \{ \textit{def}(d_2) \}), (p_1, Ø), (g, Ø)>$$
$$\dots$$
$$E_N = <(s, Ø), (d_N, \{ \textit{def}(d_N) \}), (p_1, Ø), (g, Ø)>$$

For simplicity, we assume that each definition defines the value for only one variable. Therefore, the constraint set associated with each last definition d_i in E_i is one element set *def*(d_i) that requires the value of the variable defined by d_i is to be preserved during program execution between d_i and p_1.

To proceed, the chaining approach selects one of the event sequences, e.g. E_1, and attempts to find test inputs for which it is successfully traversed. If such a test input is found, the chaining approach terminates since the test goal has been explored. If not,

during the traversal of event sequence E_1, a new problem node p_{1_1} may be encountered, e.g. between the start node s and d_1. If so, p_{1_1} is inserted into the event sequence:

$$E'_1 = <(s, \emptyset), (p_{1_1}, \emptyset), (d_1, \{ def(d_1) \}), (p_1, \emptyset), (g, \emptyset)>$$

The chaining approach again performs data dependence analysis to find the set of all last definitions for node p_{1_1}. New event sequences are generated by inserting these definitions into the temporary event sequence E'_1:

$$E_{1_1} = <(s, \emptyset), (d_{1_1}, \{ def(d_{1_1}) \}), (p_{1_1}, \emptyset), (d_1, \{ def(d_1) \}), (p_1, \emptyset), (g, \emptyset)>$$
$$E_{1_2} = <(s, \emptyset), (d_{1_2}, \{ def(d_{1_2}) \}), (p_{1_1}, \emptyset), (d_1, \{ def(d_1) \}), (p_1, \emptyset), (g, \emptyset)>$$
$$\dots$$
$$E_{1_M} = <(s, \emptyset), (d_{1_M}, \{ def(d_{1_M}) \}), (p_{1_1}, \emptyset), (d_1, \{ def(d_1) \}), (p_1, \emptyset), (g, \emptyset)>$$

Created event sequences may be organized in the form of a tree referred to as a *search tree*. The initial event sequence E_0 represents the root of the tree. Other levels of the tree are formed by event sequences created when problem nodes are encountered. Each tree node represents a possibility to unclose the test goal. Figure 4.4 shows the structure of the search tree generated during the search process.

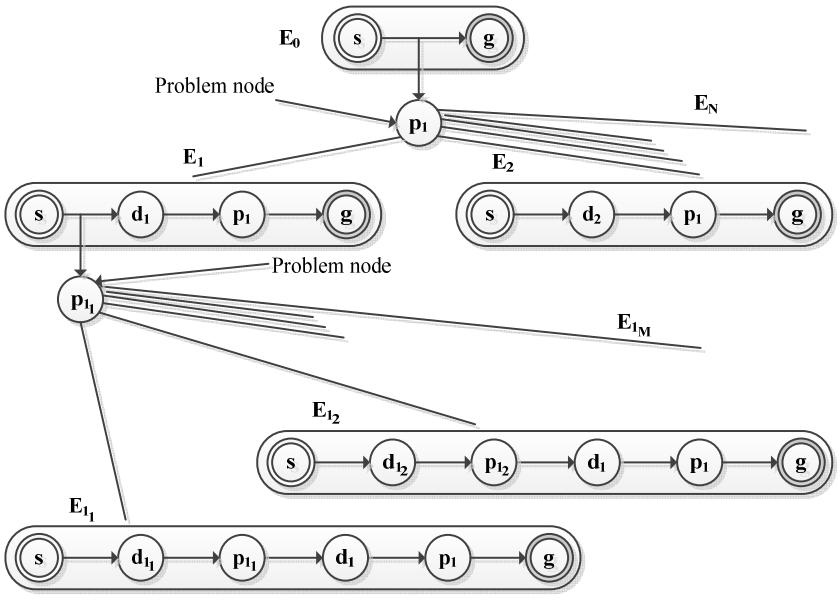

Figure 4.4: A search tree generated by the chaining approach

Figure 4.5 shows the search tree generated during exploring node 8 in the `example02` program in Figure 4.2.

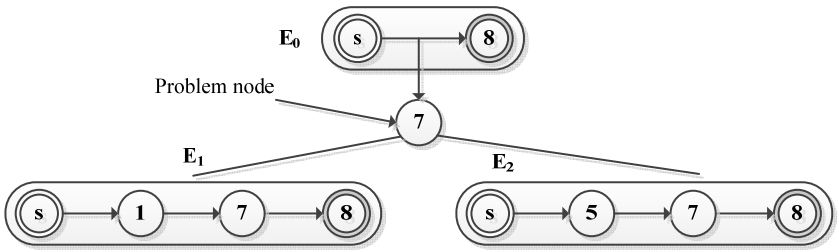

Figure 4.5: A search tree generated in exploring node 8 in the `example02` program in Figure 4.2

4.3.3 Event Sequence Generation

We now formalize the process of creating a new event sequence from an existing sequence E. Let E = <e_1, e_2, …, e_{i-1}, e_i, e_{i+1}, …, e_m> be an event sequence. Suppose the search process driven by this event sequence guides the program execution up to event e_i and a problem node p is encountered between events e_i and e_{i+1}. Let d be a last definition node of problem node p. Two events are generated, e_p = (p, Ø) and e_d = (d, *def*(d)), corresponding to the problem node and the definition. A new event sequence is now created by inserting these two events into event sequence E. Event e_p is always inserted between e_i and e_{i+1}. However, event e_d in general, may be inserted in any position between e_1 and e_p. Suppose the insertion of event e_d is between events e_k and e_{k+1}. The following sequence is then created:

$$E' = <e_1, e_2, \ldots, e_{k-1}, e_k, \mathbf{e_d}, e_{k+1}, \ldots, e_{i-1}, e_i, \mathbf{e_p}, e_{i+1}, \ldots, e_m>$$

Since new events are added to the sequence, the implication of data propagation may be violated. This requires modifications of the associated constraint sets of involved events. The update is done in the following three steps:

(1) $C_d = C_k \cup def(d)$

(2) $C_p = C_i$

(3) $\forall j, k + 1 \leq j \leq i, C_j = C_j \cup def(d)$

In the first step, the constraint set C_d of event e_d is initialized to the union of *def*(d) and the constraint set of the preceding event e_k. This modification ensures that the constraint set C_k of event e_k is preserved up to event e_{k+1} while going through the newly inserted event e_d. The second step also imposes the same requirement on event e_p by assigning C_i to its constraint set. In the final step, all constraint sets of events between e_{k+1} and e_i are modified by including a variable defined at d. By doing this, the chaining approach guarantees to propagate the effect of the definition at node d up to the problem node p.

Given this formalization, the search process when following an event sequence attempts to adjust the program execution to move from one event to another without violating the constraint set in the preceding event. This implies a systematic mechanism to propagate the effect of "all possible" data flows up to the goal structure. Unfortunately, this implication is not correct. We investigate in depth this phenomenon by assessing the sequence generation process employed in the chaining approach in the next section.

4.4 The Extended Chaining Approach

The chaining approach was utilized in this research after we observed that the control dependence information of the program may not be sufficient to guide the search process in finding test inputs to explore high complexity code [52]. By recognizing search failure may be due to data dependences, the chaining approach employs a backup strategy through the construction of event sequences which may guide the search process to propagate desired data flows to trigger the execution of test goals. The construction of event sequences is done by inserting new events which navigate the search process to take into account last definitions of variables used at problem nodes. However, last definitions *alone* might not be able to provide precise guidance toward influencing the outcome at problem nodes.

```
Node  #define THRESHOLD  10
      #define N          20
(s)   void example03 (int A[N]) {
(1)        int counter = 0;

(2)        int i = 0;
(3)        while (i < N) {
(4)             if (A[i] == 25) {
(5)                  counter ++;
                }
(6)             i ++;
           }

(7)        if (counter == THRESHOLD) {
(8)             // target
           }

(9)        return;
(e)   }
```

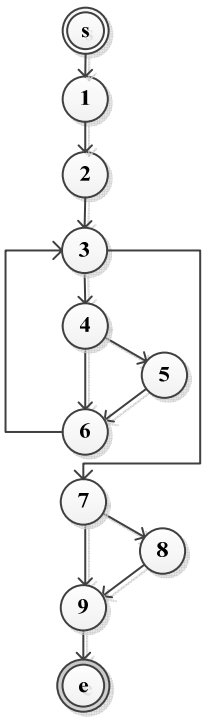

Figure 4.6: Example to illustrate limitations in the event sequence generation process of the chaining approach

4.4.1 Limitations of the Chaining Approach

Consider the `example03` program in Figure 4.6 as an example. The input to this function is an array of 20 elements. Suppose the test goal is to explore node 8, which is only executed when half of the input array elements equal 25. And now suppose that the search process fails to find such a test input to explore the test goal. Node 7 is a problem node. By performing data dependence analysis with respect to the only variable `counter` used in the conditional expression of the problem node, the chaining approach creates the following two event sequences:

$$E_1 = \langle(s, \emptyset), (1, \{\texttt{counter}\}), (7, \emptyset), (8, \emptyset)\rangle$$
$$E_2 = \langle(s, \emptyset), (5, \{\texttt{counter}\}), (7, \emptyset), (8, \emptyset)\rangle$$

Obviously, event sequence E_1 is infeasible since the value of `counter` variable being carried by E_1 is zero while the desired value to execute true branch (7, 8) is 10. Event sequence E_2 is not feasible, but requires that the `counter` variable is incremented only once. Rarely this is sufficient to ensure that a test input can be found with 9 among the remaining 19 elements equal 10 to execute the test goal. Consequently, node 7 is again a problem node. The original chaining approach is not designed to deal with this situation however; it terminates and reports node 8 could not be explored.

Furthermore, consider the `example04` program in Figure 4.7 where the test goal is to explore node 16. The execution of this node can only be triggered when all elements of both the input arrays A and B equal 25. When the search process fails to find such a test input, the chaining approach declares node 15 to be a problem node and creates the following event sequence:

$$E_1 = \langle(s, \emptyset), (14, \{\texttt{success}\}), (15, \emptyset), (16, \emptyset)\rangle$$

Intuitively, when following event sequence E_1, the search process can always propagate the only last definition at node 14 of variable `success` down to the problem node 15. However, by doing so, rarely can branch (15, 16) be executed since both `succ01` and `succ02` variables can carry `false` values. This is because there is no guidance encoded in the event sequence E_1 specifying what values `succ01` and `succ02` variables must carry to compute the value for variable `success`. As a consequence, the search process after following event sequence E_1 encounters the same problem node 3. The chaining approach terminates and reports that node 16 could not be explored.

```
Node  typedef enum {false, true} bool;
      #define N 20
(s)   bool example04 (int A[N], int B[N]) {
(1)       bool succ01 = true;
(2)       bool succ02 = true;
(3)       bool success = false;

(4)       int i = 0;
(5)       while (i < N) {
(6)           if (A[i] != 25) {
(7)               succ01 = false;
              }
(8)           i ++;
          }

(9)       int j = 0;
(10)      while (j < N) {
(11)          if (B[j] != 25) {
(12)              succ02 = false;
              }
(13)          j ++;
          }

(14)      success = succ01 && succ02;

(15)      if (success) {
(16)          // target
          }

(17)      return success;
(e)   }
```

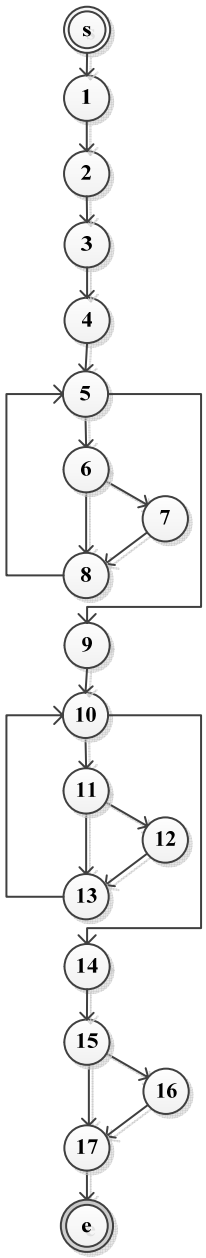

Figure 4.7: Example to illustrate limitations in the event sequence generation process of the chaining approach

The failure of the chaining approach in the two examples above originates from the following two main reasons:

- The chaining approach resolves only one level of data dependences. Obviously, if following an event sequence results in failure, the sequence should be extended to include further *transitive* data dependences to the problem node to continue exploring the test goal.
- The chaining approach takes data dependences results in isolation. It is obvious in the second example that event sequence E_1 can carry `true` value of variable

`succ01` to node 14 but the value of `succ02` can be `false`; and vice versa. To address this issue, the sequence generation process should consider *all possible* combinations of data dependences when creating new event sequences.

These limitations of the chaining approach have been addressed in the work of McMinn and Holcombe [92], who developed the *extended chaining approach*, which is detailed in the following section.

4.4.2 Extended Event Sequence Generation

The key idea behind the extended chaining approach is to take into account the effect of not only *direct* data dependences but also *indirect* data dependences in guiding the search process toward exploring a given problem node [92]. Specifically, an extension is made to the event sequence generation process to consider definitions for all variables that can potentially affect the outcome at the problem node. The extended chaining approach enables this through using the concept of *influencing sets*, which capture all variables whose definitions can either directly or indirectly influence the problem node.

The extended event sequence generation process is sketched in Figure 4.8 through the GenerateEventSequences procedure. The input to GenerateEventSequences includes event sequence E, two events e_1 and e_2, and problem node pb, or GenerateEventSequences(E, e_1, e_2, pb). The context for calling this procedure is that the search process while attempting to traverse event sequence E from event e_1 to event e_2 encountered problem node pb. The procedure performs the event sequence generation process in the following manner. Initially, for the given input problem node, the influencing set is simply the set of variables used in the conditional expression of the problem node. Paths are traversed backward from the problem node. The influencing set is adapted according to the path taken. So, starting from the current problem node pb, its initial influencing set I_{pb}, and the event $e_1 = (n_1, C_1)$ prior to the problem node event inserted into the event sequence E, the procedure invokes the CreateEventSequences procedure to traverse the control flow graph of the program under test in a backward manner. For procedure CreateEventSequences, it visits each node pn in prev_nodes, which is simply the set of program nodes connected to the current node by an outgoing edge. For each visited node pn, there are the following four possible scenarios.

Algorithm 4 GenerateEventSequences

let E be the original event sequence from which new event sequences are required

let S be a global set of search points, where a search point is a tuple sp = (sn, I, e), where sn is a program node, I is an influencing set of variables, and e = (n, C) is an event in the original event sequence E

```
procedure CreateEventSequences(In: a search point, sp = (sn, I, e = (n, C)))
1:  let prev_nodes be the set of control flow graph nodes connected to sn by an
    outgoing edge
2:  if sp ∉ S then
3:      S := S ∪ { sp }
4:      for each pn ∈ prev_nodes do
5:          if pn = n then
6:              if def(pn) ∈ I then
7:                  I := I \ { def(pn) }
8:                  I := I ∪ uses(pn)
9:              end if
10:             CreateEventSequences((pn, I, GetPreviousEvent(E, e)))
11:         else if ∀v ∈ C and v ≠ def(pn) then
12:             if ∃v ∈ I and v = def(pn) then
13:                 if Reachable(e, pn) then
14:                     CreateNewEventSequences(E, e, pn)
15:                 end if
16:                 CreateEventSequences((pn, I \ { def(pn) }, e))
17:             else
18:                 CreateEventSequences((pn, I, e))
19:             end if
20:         end if
21:     end for
22: end if
end procedure
```

def(n) returns the variable defined at program node n, or none if n is not a definition

uses(n) returns the set of variables used by program node n

Reachable(e, pn) checks if there exists a path along which node n of event e = (n, C) can reach to node pn without violation of the constraint set C

CreateNewEventSequences(E, e, pn) creates a new event sequence using the original event sequence generation process by which definition node pn is inserted into the original event sequence E after event e

GetPreviousEvent(E, e) returns the event prior to the event e in an event sequence E

Figure 4.8: Recursive procedure for generating event sequences using influencing sets in the extended chaining approach

Scenario 1 — The currently visited node pn is the same as the node n of the prior event e = (n, C) (line 5). If so, the procedure checks if pn is a definition node defining a variable in the influencing set, or *def*(pn) ∈ I. If this is the case, the influencing set is modified by removing the variable defined at pn (*def*(pn)) and by adding variables used at pn (*uses*(pn)). This is because the effect of *def*(pn) can no longer affect the problem node as the traversal passes over node pn. However, the variables used at pn can affect the problem node since they are used to compute the value assigned to *def*(pn). By doing this, the extended chaining approach enables the event sequence generation process to incorporate the effect of *transitive* data dependences when creating new event sequences. CreateEventSequences then recurses using the visited node pn, the updated influencing set, and the event prior to the input event e.

Scenario 2 — The currently visited node pn is a definition node that defines a variable in the constraint set C of the preceding event e. In this case, any path passing through node pn to the next event is not definition-clear and therefore the procedure stops going backward further from this node.

Scenario 3 — The currently visited node pn does not define any variable in the constraint set, but instead defines a variable in the current influencing set (lines 11–12). In this case, node pn presents whether a *direct* data dependence or whether an *indirect* data dependence that should be propagated up to the problem node. The procedure then checks if there exists a definition-clear path with respect to the constraint set C from event e to node pn. If so, a new event sequence is generated using the original event sequence generation process as described in section 4.3.3 for the definition node pn. The procedure then recurses using the new influencing set $I' = I \setminus \{ def(pn) \}$.

Scenario 4 — In this scenario, the procedure simply recurses using the currently visited node pn along with the current influencing set I and event e (line 18).

Note that during the traversal of program nodes in the extended event sequence generation process, a global data structure of search points is used to ensure that the traversal terminates when traversing cyclic paths in the program.

Intuitively, the recursive algorithm to perform the event sequence generation process presented above is basically similar to the program slicing algorithm proposed in the work of Weiser [131] in 1981. The key difference between the two techniques is that the

event sequence generation process does only consider data dependence information while Weiser's slicing algorithm takes into account both data and control dependences. We now demonstrate how the use of influencing sets and the extended chaining approach can tackle limitations encountered with the original chaining approach.

Consider again the `example04` program in Figure 4.7 where the search process traversing event sequence E_1 failed to explore the test goal node 16 and encountered again the problem node 15:

$$E_1 = \langle(s, \emptyset), (14, \{\texttt{success}\}), (15, \emptyset), (16, \emptyset)\rangle$$

For now, in the extended event sequence generation process, the influencing set is $I = \{\texttt{success}\}$. In traversing the control flow graph backward from the current problem node, node 14 is encountered. This corresponds to the first scenario in the algorithm since an event of node 14 appears in event sequence E_1. The influencing set is adapted by removing the definition variable `success`:

$$\begin{aligned} I &\leftarrow I \setminus \{\, def(14) \,\} \\ &\leftarrow \{\texttt{success}\} \setminus \{\texttt{success}\} \\ &\leftarrow \emptyset \end{aligned}$$

resulting in an empty set, and by adding used variables `succ01` and `succ02`:

$$\begin{aligned} I &\leftarrow I \cup uses(14) \\ &\leftarrow \emptyset \cup \{\texttt{succ01}, \texttt{succ02}\} \\ &\leftarrow \{\texttt{succ01}, \texttt{succ02}\} \end{aligned}$$

With the presence of variables `succ01` and `succ02` in the influencing set, the event sequence generation process is forced to take into account their last definitions at nodes 1, 2, 7, and 12 as specified in scenario 3. As a result, the following four event sequences are created accordingly:

$$E_{1_1} = \langle(s, \emptyset), (1, \{\texttt{succ01}\}), (14, \{\texttt{success}\}), (15, \emptyset), (16, \emptyset)\rangle$$
$$E_{1_2} = \langle(s, \emptyset), (2, \{\texttt{succ02}\}), (14, \{\texttt{success}\}), (15, \emptyset), (16, \emptyset)\rangle$$
$$E_{1_3} = \langle(s, \emptyset), (7, \{\texttt{succ01}\}), (14, \{\texttt{success}\}), (15, \emptyset), (16, \emptyset)\rangle$$
$$E_{1_4} = \langle(s, \emptyset), (12, \{\texttt{succ02}\}), (14, \{\texttt{success}\}), (15, \emptyset), (16, \emptyset)\rangle$$

Event sequences E_{1_3} and E_{1_4} are infeasible. Event sequences E_{1_1} and E_{1_2} are unlikely to provide sufficient guidance toward exploring node 16. This is because when traversing, for example, event sequence E_{1_1}, the value of `succ01` variable is `true` but variable `succ02` can be `false`, resulting in an unintended false branch (15, 17). That is, node 15 remains problematic and the following event sequence will be generated when calling the extended event sequence generation process on event sequence E_{1_1} with the problem node 15:

<(s, Ø), (1, {succ01}), (2, {succ01, succ02}), (14, {success}), (15, Ø), (16, Ø)>

This event sequence guides the search process to propagate the desired `true` value of both variables `succ01` and `succ02` defined at nodes 1 and 2 up to node 14 for evaluating the value for `success` variable. The value of `success` is evaluated to `true` providing the intended value to trigger the execution of the true branch (15, 16).

Consider again the `example03` program in Figure 4.6 in which we evaluated how the extended event sequence generation process can help to unroll the loop for uncovering the test goal node 8. The search process when traversing event sequence E_2 failed to explore the test goal and the problem node 7 was encountered again:

$$E_2 = \langle(s, Ø), (5, \{counter\}), (7, Ø), (8, Ø)\rangle$$

The influencing set is I = {`counter`}. In traversing the control flow graph backward from the problem node 7, node 5 of event (5, {`counter`}) is encountered. An update to the influencing set occurs by removing the variables defined at node 5 and adding the variables used. This results in the same influencing set {`counter`}. The presence of variable `counter` after traversing node 5 forces the consideration of its last definitions at nodes 1 and 5. The following event sequences are generated to continue guiding the search process:

$$E_{2_1} = \langle(s, Ø), (1, \{counter\}), (5, \{counter\}), (7, Ø), (8, Ø)\rangle$$
$$E_{2_2} = \langle(s, Ø), (5, \{counter\}), (5, \{counter\}), (7, Ø), (8, Ø)\rangle$$

Event sequence E_{2_2} directs the search process to unroll the loop two times, giving a better value of `counter` variable to satisfy the condition `counter == THRESHOLD`. This process is repeated until there are `THRESHOLD` (or 10) times the presence of event (5, {`counter`}) in an event sequence to execute node 8.

4.5 Goal-Oriented Dynamic Test Generation

In this section, we present our proposed search algorithm to carry out goal-oriented test input generation. The proposed algorithm, which we call GUIDER, employs dynamic symbolic execution to perform test input generation and makes use of the chaining approach to guide the path exploration process toward effectively and efficiently exploring a given test goal.

The entry point of the algorithm is given in Figure 4.9. It takes as input a program under test P, a test goal g, and a testing limit *limit*. The output is a test input t such that the execution of P with t executes g, or *null* implying such t was not found, or *limit* was expired.

Algorithm 5 The Chaining Guided Search Algorithm (GUIDER)

Input : Program P, Test goal g, Testing limit *limit*
Output : Test input t (or *null*)

```
1:  E0 := CreateInitialSequence(g)
2:  t0 := GenerateRandomInput(P)
3:  p0 := ExecuteProgram(P, t0)
4:  worklist := { (E0, p0) }
5:  while worklist is not empty and limit is not expired do
6:      (s, p) := SelectEventSequence(worklist)
7:      (explored, t) := ExploreEventSequence(s, p)
8:      if explored then
9:          return t
10:     end if
11:     RemoveEventSequence((s, p), worklist)
12: end while
13: return null
```

Figure 4.9: A goal-oriented dynamic test generation algorithm guided by the chaining approach and based on dynamic symbolic execution

The search algorithm uses *worklist* to keep all generated event sequences during the search process. Associated with each event sequence is a program execution, which is used to perform path exploration toward traversing the event sequence. Note that traversing an event sequence completely implies that the test goal g was executed since the last event of all event sequences always refers to g (see sections 4.3 and 4.4). The event sequence traversal is done in each iteration of the **while** loop by calling procedure ExploreEventSequence (Figure 4.10). This process is repeated until either an input was found to explore the test goal g or the testing limit *limit* was expired.

```
procedure ExploreEventSequence(E, p)
14: e1 := E[1]
15: e2 := E[2]
16: PP := GetProgramPath(p)
17: s := PP[1]
18: while true do
19:     s := PP[IndexOf(s) + 1]
20:     if s = e2→n then
31:         if e2 = E[end] then
32:             return (true, π(p))
33:         end if
34:         e1 := e2
35:         e2 := E[IndexOf(e2) + 1]
36:     else if s violated e1→C then
37:         (adjusted, p′, s′) := AdjustWhenViolated(e1, e2, E, s, p)
38:         if adjusted then
39:             p := p′
40:             s := s′
41:         else
42:             return (false, null)
43:         end if
44:     else if s is a branch statement then
45:         b := GetAlternativeBranch(s)
46:         if b has minimal distance to e2→n and s is a symbolic predicate then
47:             (adjusted, p′) := SolveAtBranch(s, p)
48:             if adjusted then
49:                 p := p′
50:                 s := GetConditionalStmt(b)
51:                 continue
52:             end if
53:         end if
54:         if s cannot reach e2→n then
55:             RefineEventSequence(e1, e2, E, s, p)
56:             return (false, null)
57:         end if
58:     end if
59: end while
end procedure
```

Figure 4.10: Procedure ExploreEventSequence in the goal-oriented dynamic test generation algorithm

The core functionality of the algorithm lies in the ExploreEventSequence procedure. Put simply, this procedure performs a *pattern concretization algorithm*, where the input event sequence E can be considered to be a target pattern and the input execution p is to be adjusted in order to concretize E. Stated formally:

— Given an event sequence E = <e_1, e_2, e_3, …, e_m> and a program execution p, the goal is to find a program execution p′ on which E is concretized.

This problem can be further reduced to the problem of concretizing every two *adjacent* events of event sequence E. To do this, we use e_1 and e_2 to capture every two adjacent events on E (lines 14–15) and s to iterate over every executed statement on the executed program path PP to inspect a concretization of e_1 and e_2. An invariant maintained during the concretization inspection is that event e_1 has already been concretized (or node $e_1 \rightarrow n$ was found), and the goal is to reach event e_2 (or to find node $e_2 \rightarrow n$) without modifying any variable in the constraint set $e_1 \rightarrow C$. Note that this invariant is satisfied in the beginning as e_1 points to the first event of E, which is actually the program entry (section 4.3), and the path iteration is started at the statement right after the program entry (lines 17 and 19). The preservation of the constraint set C of event e_1 is to propagate data definitions up to the target structure. Now, going down along the executed path, we inspect every executed statement s and consider the following four possible scenarios.

Scenario 1 — The target event e_2 is discovered (line 20). This is found by checking if the currently inspecting statement s is the program node of e_2. If so, we update e_1 and e_2 to the next two events of E to continue the concretization process (lines 34–35). In case e_2 is the last event of E, the concretization of E has been accomplished on the executed path PP. The algorithm terminates by returning the input executing p, i.e. $\pi(p)$ (line 32).

Scenario 2 — The currently inspecting statement s violates the constraint set C of event e_1 (line 36). This is found by checking if s is a definition statement that redefines any variable in C. If so, the implication of data propagation encoded in constraint C of event e_1 is no longer valid. In this case, we attempt to adjust the current program execution p to avoid the execution of this violating statement s through calling AdjustWhenViolated procedure (Figure 4.11).

The AdjustWhenViolated procedure takes as input two events e_1 and e_2, sequence E, violating statement vs, and program execution p. It goes backward along the executed path PP, starting from the violating statement vs to the statement where event e_1 was discovered, and examines at each statement encountered to perform an execution adjustment. Specifically, for each statement b, it checks the following four conditions:

1. if b is a branch statement and
2. if b is a symbolic predicate and
3. if the violating statement vs is *transitively* control dependent on branch b and
4. if the *alternative* branch of b can reach event e_2.

```
procedure AdjustWhenViolated(e1, e2, E, vs, p)
60: // Phase 1: Adjust the execution to avoid the violation
61: PP := GetProgramPath(p)
62: for each b in range ( vs, e1→n ) on PP do
63:     if b is a branch statement and
64:         b is a symbolic predicate and
65:         vs is transitively control dependent on b and
66:         alternative branch of b can reach e2→n then
67:           (satisfied, p′) := SolveAtBranch(b, p)
68:           if satisfied then
69:               c := GetConditionalStmt(b)
70:               return (true, p′, c)
71:           end if
72:     end if
73: end for
74: // Phase 2: Refine event sequence as the adjustment failed
75: RefineEventSequence(e1, e2, E, vs, p)
76: return (false, null, null)
end procedure
```

Figure 4.11: Procedure AdjustWhenViolated in the goal-oriented dynamic test generation algorithm

The first two conditions, (1) and (2), are to ensure that b can be flipped, and the last two conditions, (3) and (4), are to ensure that flipping of b to its alternative branch avoids the execution of vs and reaches event e_2. If these four conditions together are satisfied, then the flipping is computed by invoking the SolveAtBranch procedure (Figure 4.13). The satisfiability of the flipping yields a new program execution p' on which the sequence concretization can safely proceed downward from the conditional statement c of branch b. The soundness of doing so is guaranteed by two properties:

1. the executed paths PP' of p' and PP of p match identically from the program entry up to statement c and
2. the flipping is restricted to (branch) statements down below the statement where event e_1 was discovered.

These two properties ensure that the sequence concretization result up to c is preserved. In case the path adjustment failed, the RefineEventSequence procedure (Figure 4.12) is invoked to refine the current event sequence E (line 75). We describe the sequence refinement procedure below.

Scenario 3 — The currently inspecting statement s is a branch statement and its alternative branch b has a minimal distance to reach event e_2 (lines 44–46). This scenario results from the observation that if the current execution p can change the

control flow to execute this minimal distance branch b, it may potentially reach event e_2 *quickly*. For this, the algorithm checks if s is also a symbolic predicate and hence can be flipped. If so, the flipping is performed by invoking the SolveAtBranch procedure to change the execution of p from s to b (lines 47–53). This scenario represents our algorithm's attempt to optimize path exploration.

Scenario 4 — The currently inspecting statement s is a branch statement and s cannot reach event e_2 (line 54). This is determined by confirming that there does not exist a program path from s to node n of event e_2 in the static control flow graph. In this case, procedure RefineEventSequence is invoked to refine the current event sequence E.

```
procedure RefineEventSequence(e1, e2, E, s, p)
77: PP := GetProgramPath(p)
78: for each b in range ( e1→n, s ] on PP do
79:     if b is a branch statement then
80:         b′ := GetAlternativeBranch(b)
81:         if e2→n is transitively control dependent on b′ and
82:             b′ has minimal distance to e2→n then
83:             c := GetConditionalStmt(b′)
84:             S := CreateEventSequences(E, e1, e2, c)
85:             worklist ← worklist ∪ { (E′, p) | E′ ∈ S }
86:         end if
87:     end if
88: end for
end procedure
```

Figure 4.12: Procedure RefineEventSequence in the goal-oriented dynamic test generation algorithm

Next, we illustrate how the sequence refinement procedure RefineEventSequence works to refine a given event sequence. The input to RefineEventSequence includes two events e_1 and e_2, event sequence E, statement s, and program execution p. The context of calling this procedure is that the sequence concretization process while attempting to reach event e_2 encountered statement s where:

- s violated the constraint C of event e_1 and the algorithm failed to adjust the program execution p in order to avoid the execution of s (scenario 2), or
- the program execution p if following s can no longer reach event e_2 (scenario 4).

The main functionality of the RefineEventSequence procedure is to identify problem nodes causing the failure of concretizing sequence E on the executed path PP of execution p. To do this, it inspects all branch statements from node n of event e_1 down

to statement s, and at each branch b checks its alternative branch b′ the following conditions:

1. if event e_2 is *transitively* control dependent on b′ and
2. if event b′ has a minimal distance to event e_2.

The first condition is to ensure that changing the execution of p from b to b′ can eventually reach e_2. The second condition takes into consideration that the execution of b′ can potentially reach event e_2 quickly. If these two conditions together are satisfied, then the conditional statement c of b′ is considered to be a problem node used to refine sequence E. The GenerateEventSequences procedure (Figure 4.8) is then invoked to incorporate direct (and indirect) data dependences of c into sequence E to create more refined event sequences. These created sequences are associated with the current execution p before being added into *worklist*.

Finally, the SolveAtBranch procedure is given in Figure 4.13. This procedure attempts to change the input program execution p from branch b to its alternative branch b′.

```
procedure SolveAtBranch(b, p)
89: φ := GetPathConditionUpTo(p, b)
90: suppose φ = σ1 ∧ σ2 ∧ ... ∧ σi−1 ∧ σi
91: φ′ := σ1 ∧ σ2 ∧ ... ∧ σi−1 ∧ ¬σi
92: (satisfied, t) := SolveConstraintSystem(φ′)
93: if satisfied then
94:     p′ := ExecuteProgram(P, t)
95:     return (true, p′)
96: end if
97: return (false, null)
end procedure
```

Figure 4.13: Procedure SolveAtBranch in the goal-oriented dynamic test generation algorithm

4.6 Summary

In this chapter, the problem of generating test inputs to target a specific code element in a program was presented. Formally, this problem is referred to as the reachability problem. In general, it is undecidable problem. A number of useful applications of the reachability problem in several stages of the software development life cycle have attracted much attention in research community. A primary objective of this research project is to explore techniques to effectively and efficiently improve the solving the reachability problem. Dynamic symbolic execution has been shown to be an effective

technique to automate test input generation of software testing. However, this technique suffers greatly from the combinatorial explosion of the path space. Because testing resources are almost always limited, much research has extensively investigated search algorithms to optimize path exploration in dynamic symbolic execution. In the context of exploring search algorithms to improve the efficiency of path exploration, control and data flow information can potentially guide the search process. While control flow information can navigate the search process to quickly reach the test goal, it cannot resolve data dependences, which is necessary to trigger the execution of the test goal. The employment of data flow information to guide the search process was suggested by the chaining approach [52] and was later enhanced in the extended chaining approach [92]. The intertwinement of dynamic symbolic execution with the chaining approach is proposed in this research project to improve solving the reachability problem.

This chapter focused on the development of a goal-oriented test input generation approach. Section 4.1 provided background to formally present the chaining approach. The significance of developing a goal-oriented approach of test input generation was justified in section 4.2. For this, we emphasized the need to deal with the combinatorial explosion of the path space faced by dynamic symbolic execution. We noted that the efficiency of performing dynamic symbolic execution can be affected by a number of factors [114]; in this project, we concentrate only on the path space explosion problem. To address the insufficiency of control flow guidance, the employment of data flow information was utilized.

Section 4.3 described the chaining approach and the need for a search mechanism that can direct the search process to effectively and efficiently target a test goal was emphasized. The key idea behind the chaining approach is the encoding of data dependences in forms of event sequences and guiding the search process to influence the execution of the test goal. In section 4.4, the extended chaining approach was presented, which takes into account transitive data dependences when creating event sequences.

A goal-oriented test input generation approach was described in section 4.5. The approach intertwined dynamic symbolic execution to automate test input generation and the chaining approach to identify and solve data dependences for exploring test goals. The chaining approach's ability to precisely focus on executing the test goal can positively impact the path space explosion problem facing dynamic symbolic execution.

It is important to notice that the goal-oriented test input generation approach proposed in this thesis differentiates itself from the work of McMinn and Holcombe [92] in several aspects. Specifically, McMinn and Holcombe extended the chaining approach and then combined with search-based testing for structural coverage testing. In our proposal, we employed the extended chaining approach to form a search mechanism for carrying out path exploration of dynamic symbolic execution. Practically, generating test inputs using meta-heuristic search techniques of search-based testing is far from being capable of testing real world software. Dynamic symbolic execution has been used to test practical software with millions of lines of code [66]. Additionally, while the work of McMinn and Holcombe is limited to structural coverage testing, our proposed approach is to be applied to security vulnerability testing which is one of the most important issues challenging today's software testing.

In the next two chapters, we describe the two applications of our proposed goal-oriented test input generation approach for structural coverage testing and security vulnerability testing.

Chapter 5

Structural Program Coverage

High structural coverage achievements have been long advocated as a convenient way to measure the adequacy of software testing [20], [56], [117], [132]. Over the last three decades, researchers have defined several testing criteria based on structural coverage of the program under test to improve confidence in software quality and reliability upon deployment [4]. Meanwhile, considerable research effort has also attempted to develop techniques to improve the efficiency of software testing via structural coverage enhancements. A focus of this effort has been to develop effective and efficient techniques to automate the process of generating test inputs for covering many coverage elements of the program under test [29], [52], [87], [104], [112].

With the recognition that a structural coverage element is actually a code element of the program's source code, the testing approach can be reduced to the task of generating test inputs to cover a specific code element. In this context, our goal-oriented testing approach proposed in the previous chapter can be utilized to carry out structural coverage testing. A distinguishing feature of our proposed approach is the ability to guide path exploration of dynamic symbolic execution to propagate data flows down to the goal structural to trigger the execution of high complexity code. This is a significant

improvement over existing techniques to deal with the combinatorial explosion of the path space for high structural coverage achievements.

This chapter is organized as follows. Section 5.1 covers theoretical concepts of structural coverage criteria. Section 5.2 provides a literature review of automated test input generation techniques for structural coverage testing. We mainly focus on assessing the strengths and shortcomings of techniques based on dynamic symbolic execution to clarify the context of our research proposal. In section 5.3, our proposed structural coverage testing framework SCT is presented. We explain how GUIDER, the chaining guided search algorithm proposed in the previous chapter, can be used to form the SCT framework and also cover its implementation details. In section 5.4, we present an evaluation of the effectiveness of SCT in relation to structural coverage improvements. We describe the test subjects selected, the evaluation methodology, the experimental results, and provide a discussion. The capability of SCT is assessed in comparison with popular test input generation approaches, and its ability to achieve high structural coverage results and optimize path exploration within the constraint of testing limits is highlighted. Finally, a summary of the chapter is given in section 5.5.

5.1 Overview

Software testing is potentially endless. To ensure the correctness of the software under test, one has to test and verify the software against all the possible input values. Unfortunately, complete testing is infeasible. Consider testing a program of adding two integer input values of 32-bits, for example. It yields a set of $2^{32} * 2^{32} = 2^{64}$ distinct test cases. Now if test cases were performed at a rate of thousands of test cases per second, it would take hundreds of years to accomplish exhaustive testing for this program, and for real world software applications the input space goes far beyond this simple program.

In the practice of software development, software testing is a profit-driven model. It is a practical trade-off between project budget, time, and quality. Realistically, at some point, the software testing process has to be terminated and the software application can then be delivered to customers. In the understanding of the incompleteness of software testing, measurement must be developed in accordance with software quality factors to perform the testing process and to access the efficiency of software testing. This project

has attracted considerable research effort to investigate and propose sufficient measures for the termination of the software testing process. A major research focus is the definition of criteria determining what constitutes an adequate test. Over the last three decades, a great number of such criteria have been proposed and developed by the research community [4]. Much effort has also been directed at providing support for the use of one criterion over another [4]. In fact, the benefit of including a test adequacy criterion in the software testing process is two-fold:

- An adequacy criterion can be considered to be a stopping rule for determining whether sufficient testing has been carried out that it can be terminated.
- Test adequacy criteria provide measurements of test quality where a degree of adequacy associated with a test set can give a level of confidence about the correctness of the software under test.

These two important benefits have led to empirical studies advocating the adoption of test adequacy criteria into industrial software development practice [56], [132]. Internationally accepted testing standards such as the British Standards Institute [20] and the Radio Technical Commission for Aeronautics [117] have adopted standardized test adequacy measurements for software quality and reliability. In the software testing research community, test adequacy criteria continue to be an attractive research topic. Attempts have been made to provide a comprehensive understanding of the important usages of adequacy criteria with respect to various software quality assessments indispensable to the software testing discipline [4], [145].

This research project primarily concentrates on structural coverage criteria, a simple but commonly adopted set of adequacy criteria among the many test adequacy criteria proposed in the practice of software testing [20], [56], [117], [132]. Structural coverage is a quantitative measure of the degree to which the source code of the program under test is exercised during the software testing process. A high coverage degree implies the program is thoroughly tested and has a lower chance of containing software defects than a program with low structural coverage. An important objective of this research project is therefore to explore automated techniques to improve structural coverage achievements for the program under test.

5.1.1 Structural Coverage Criteria

A number of metrics have been proposed in the software testing literature to calculate structural coverage with varying complexities [4]. Each coverage metric imposes a specific assessment of the adequacy of testing for the program under test. Often, the complexity associated with a particular coverage metric dictates which coverage metrics are used by the software industry. With sizeable and complex software applications, simple coverage metrics are preferred since the adequacy of the testing process can be measured in relation to the project budget, scope, and time constraints [20], [117]. The remainder of this section covers some of the test adequacy criteria based on structural coverage.

Statement coverage. In the practice of software testing, test engineers are required to design test cases to execute and check the program under test against given testing requirements. It is often desirable to develop a set of test cases with which every statement in the program is executed at least once. The requirement of executing all statements in the program under test is an adequacy criterion. A test set satisfying this requirement is considered to be adequate according to the statement coverage criterion.

In reality, only a fragment of the source code can be covered during testing due to the complexity of the program under test. Therefore, the percentage of executed statements is calculated to indicate how adequately the software testing process has been performed. The percentage of the statements exercised by testing is a measurement of the adequacy. Statement coverage is one of the most widely adopted test adequacy measurements in the practice of software testing due mainly to its simplicity and affordability. Adopting the statement coverage criterion into the software testing process can be justified by the observation that the higher the number of statements that are not executed and checked, the lower the quality and reliability of software is. This is because unexercised statements can harbour serious software defects.

Branch coverage. The branch coverage criterion requires that all true and false branches in the program under test must be exercised during the software testing process. The percentage of the branches exercised during software testing is a measurement of test adequacy. The branch coverage is much stronger than the statement coverage criterion because having achieved the latter does imply the former has been satisfied.

Path coverage. The path coverage criterion is the strongest test adequacy criterion and subsumes all the other criteria. It requires that all the execution paths from the program's entry to its exit are executed during the software testing process.

The adoption of a structural coverage criterion in the software testing process specifies a particular software testing requirement, and thus determines test cases to satisfy the requirement. It also designates information to measure and estimate the process of software testing. These features are essential in guiding the underlying testing method toward determining the adequacy of software testing.

5.1.2 Structural Coverage Testing

When attempting to achieve a structural coverage criterion, the structure of the program under test is modelled to direct the underlying testing method. Normally, the program is presented in the form of a control flow graph (section 4.1). Based on this graph, the testing method is able to observe the coverage result being achieved and then to navigate testing toward accomplishing the structural coverage criterion being attempted. A testing method should be designed to perform software testing for a given structural coverage criterion with the following basic steps:

1. Given a test case, execute the program under test on this test case. Then capture newly executed code elements and update the structural coverage result.
2. Identify unexercised code elements by using the currently updated coverage result and the control flow graph of the program.
3. Design test cases to execute some of the unexercised code elements as identified in the second step and then return to the first step.

This testing procedure can be repeated until one of the following two conditions is reached. The first stopping condition is if the structural coverage criterion is satisfied. This implies that the testing procedure has been able to explore all the structural elements required by the specified test adequacy criterion. This is the desired case since the testing method can ensure the adequacy of testing for the program under test. The second stopping condition is if the testing time limit is no longer valid. In this case, the structural coverage criterion has not been reached but the testing method is to be terminated. The currently achieved structural coverage result is returned to indicate how adequately the testing method has been performed.

The notion of control flow graph is fundamental to the quantitative measuring of the adequacy of software testing with respect to structural coverage criteria. It precisely captures the structure of the source code and importantly provides an abstract view by abstracting away the detail of the program under consideration. In the software testing literature, the control flow graph is used to formally define test adequacy criteria as well as to effectively estimate the adequacy of a set of test cases against the structural coverage criterion that the testing method is attempting to achieve. The execution of the program can be modelled as a traversal in the control flow graph. Every execution is a path in the control flow graph starting from the start node to the exit node. Such a path is used to determine which nodes and edges in the control flow graph have been traversed and thence to determine which statements and branches in the program under test have been executed. This is easy to justify because in the control flow graph model nodes correspond to statements while edges correspond to branches. Now, formal definitions of structural coverage criteria can be characterized in terms of graph models and test cases.

Definition 5.1. Statement Coverage Criterion. *A set S of test cases satisfies the statement coverage criterion if and only if, for all nodes n in the control flow graph, there is at least one test case s in S such that the corresponding execution path of s traverses n.*

The statement coverage criterion is a basic requirement of test adequacy; it simply requires that all the statements in the program under test are executed by test executions. The adequacy of a specific set of test cases with respect to the statement coverage criterion is determined by the percentage of the graph nodes traversed during testing. Consider the C program `example01` in Figure 5.1, for example. It consists of seven statements of which two are conditional. Correspondingly, in the control flow graph of this program there are seven nodes and four edges. Now consider the test sets S_1 = { (0, 0) } and S_2 = { (10, 10) }. The execution of S_1 can cover nodes 1, 2, 3, 4, 5, 6 and 7; or it can cover all the nodes in the control flow graph. In other words, test set S_1 achieves 100% of the statement coverage criterion for the `example01` program.

The execution of S_2 can cover nodes 1, 2, 3, 5 and 7; it misses nodes 4 and 6. The adequacy of S_2 with respect to the statement coverage criterion of the program is therefore about 71%.

```
Node  typedef enum {false, true} bool;
(s)   bool example01 (int r1, int r2) {

(1)       bool error1 = false;
(2)       bool error2 = false;

(3)       if (r1 == 0)
(4)           error1 = true;

(5)       if (r2 == 0)
(6)           error2 = true;

(7)       return (error1 && error2);

(e)   }
```

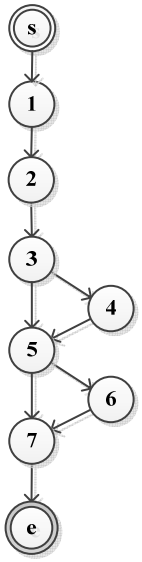

Figure 5.1: A C program and its control flow graph to illustrate structural coverage criteria

Definition 5.2. Branch Coverage Criterion. *A set S of test cases satisfies the branch coverage criterion if and only if, for all edges e in the control flow graph, there is at least one test case s in S such that the corresponding execution path of s traverses e.*

According to definition 5.2, the adequacy of a set of test cases with respect to the branch coverage criterion is measured in terms of the percentage of the graph edges traversed during testing. It is obvious that the branch coverage criterion is stronger than the statement coverage because if all edges in the control flow graph are covered, all nodes are necessarily covered. Therefore, a test set satisfying the branch coverage criterion must also satisfy the statement coverage criterion. Now consider again the `example01` program in Figure 5.1 with test sets S_1 = { (0, 0) } and S_2 = { (0, 0), (10, 10) }. For S_1, it satisfies the statement coverage criterion because it covers all the nodes in the control flow graph. However, of the four edges, S_1 only covers two edges (3, 4) and (5, 6) and therefore achieves 50% of the branch coverage criterion for the `example01` program. The adequacy of branch coverage cannot be guaranteed by the test set S_1 since it misses edges (3, 5) and (5, 7). The presence of test case (10, 10) in test set S_2 makes sure that these missing edges are covered, so that the test set S_2 achieves 100% of branch coverage for the `example01` program.

Definition 5.3. Path Coverage Criterion. *A set S of test cases satisfies the path coverage criterion if and only if S traverses all execution paths from the start node to the end node in the control flow graph.*

The path coverage criterion is the strongest criterion among test adequacy criteria, including structural coverage criteria. A test set satisfies the path coverage criterion if it traverses all the feasible execution paths in the program under test. The satisfiability of path coverage implies those of both statement and branch coverage. For example, consider test set S_1 = { (0, 0), (10, 10) } for the `example01` program in Figure 5.1. It traverses two paths, P_1 = <1, 2, 3, 4, 5, 6, 7> and P_2 = <1, 2, 3, 5, 7>. Obviously, S_1 satisfies both the statement coverage criterion and the branch coverage. However, it does not satisfy the path coverage criterion. In fact, it misses the following two paths, P_3 = <1, 2, 3, 4, 5, 7> and P_4 = <1, 2, 3, 5, 6, 7> out of the four feasible program paths. These missing paths can be covered by test cases (0, 10) and (10, 0). As a result, the test set S_2 = { (0, 0), (10, 10), (0, 10), (10, 0) } can ensure the path coverage criterion for the `example01` program.

In the context of testing real world software applications, the path coverage criterion is too strong to be practically useful. This is because there can be an infinite number of feasible paths in a program. According to the second stopping condition of the underlying testing method as outlined above, testing must be terminated after a finite period of time. Therefore, the testing method is able to explore only part of the program path space. For the statement and branch coverage criteria, completely achieving any of these coverage criteria is unfortunately infeasible. This is due to the presence of unreachable code elements in the program under test. More precisely, for these unreachable code elements there does not exist any test input to execute them. The overall objective of software testing is therefore to achieve high structural coverage within the given testing time limit. In software testing practice, the testing methods adopted usually only focus on simple structural coverage criteria such as statement or branch coverage [20], [56], [117], [132]. Developing effective and efficient methods to automate structural coverage testing is thus essential to avoid the expense and unreliability of manual testing. In the next section, we survey the automated techniques proposed in the literature for improving structural coverage to fully develop the context of this research project.

5.2 Literature Review

Test adequacy criteria are one of the most powerful testing tools for test engineers as they specifies specific testing requirements and hence enable engineers to determine

what test inputs need to be designed during software testing. They therefore help ensure high quality and reliability. Test adequacy criteria also provide an indication of when software testing should be terminated.

In the context of structural coverage testing, the software testing requirement is simply to achieve a desired structural coverage criterion and the underlying testing method generates test inputs to satisfy this requirement. The result of testing is quantitatively indicated through the percentage of the structural coverage achieved. Obviously, test engineers can manually design test inputs to accomplish this task. However, a human-centric approach to test input generation is tiresome, expensive, and unreliable. Completely automating the process of generating test inputs for structural coverage testing has therefore become a focal point for researchers. An automated testing approach could significantly reduce the expensive cost of manual software testing and also alleviate the unreliability of human intervention.

Over the last three decades, considerable research effort has developed techniques to support the process of software testing. A number of techniques have been proposed, including random testing [3], symbolic execution [112], search-based testing [87], the chaining approach [52], and dynamic symbolic execution [29]. In the context of automatically generating test inputs to achieve the adequacy of structural coverage criteria, random testing may not be effective because its capability of generating test inputs to uncover corner case branches can be extremely small [104]. Symbolic execution offers an effectively automated test input generation mechanism to systematically explore all feasible program paths but the scalability of this technique is limited due to complex constraints, data structures, and native calls of real world software systems [112]. In search-based testing [87] and the chaining approach [52], the process of generating test inputs incorporates meta-heuristic search algorithms. However, these algorithms are performed largely randomly.

Dynamic symbolic execution has been shown to be an effective technique for automated test input generation. The fundamental scalability issue facing dynamic symbolic execution however is the combinatorial explosion of the path space. Since it was proposed in the work of Godefroid *et al*. [62] and Cadar *et al.* [30] in 2005, the technique has attracted a large amount of attention, with researchers attempting to expand its applicability to testing real world software applications. The research focus

has been primarily to explore techniques to improve the efficiency of path exploration in dynamic symbolic execution. In the remainder of this section, we provide an in-depth literature review of dynamic symbolic execution-based techniques and evaluate their strengths and shortcomings in dealing with the path space explosion problem for structural coverage testing.

As noted above, a significant scalability challenge when applying dynamic symbolic execution is how to handle the combinatorial explosion of the path space. Approaches in favour of depth-first explorations such as DART [62] and CUTE [124] deeply widen the program path space but lack the ability to forward the execution to further unexplored control flow points. These approaches when executed against large programs in finite time often end up with only small regions of the code explored and fail to uncover errors in the unexplored code.

Various novel techniques have been proposed to deal with the path explosion problem, including abstraction [5], compositional [58], and parallel [126] techniques. The aim is to prune the path space to be systematically explored [5], [58] or to enlarge path exploration by exploiting the increased availability of computational power [126]. Other research explores practical trade-offs by developing search algorithms to improve the efficiency of path exploration over the path space. Most proposed algorithms focus on achieving high statement and branch coverage. PEX [128] is an automated structural testing tool developed at Microsoft Research that integrates a rich set of basic search strategies and gives a fair choice among them. While the integration does help improve code coverage by attempting different program control flows, exploring code elements may require specific guidance of control and data dependencies. FITNEX [141] makes PEX more guided by using fitness functions to measure the improvement of path exploration. The main obstacle of this approach is the flag problem [16] — when fitness functions face a flat fitness landscape they can give no guidance to the search process. Flags, unfortunately, are widely used in real world programs [16], [30], [35].

KLEE [26] is open-source and has been used by a variety of researchers in academia and industry. Like PEX, KLEE also implements a number of search heuristics and activates each heuristic in a round robin fashion for high coverage. Baluda *et al.* [12] showed however that when the executability of code elements requires data dependences computed inside loop or nested loop structures, KLEE achieves very poor coverage.

CREST [19] is an extensible platform for building and experimenting with search heuristics for achieving high structural coverage. Among the heuristics implemented in CREST, *control-flow graph directed search* (or CFGDIRECTED) is shown to be more effective than the others by experimental data. This search strategy leverages the static control flow of the program to guide the search down short static paths to unexplored code. Theoretically, the control flow guidance may be imprecise since the execution of code elements may require data dependences going beyond static short paths and/or being calculated in dynamic paths. Practically, CFGDIRECTED goes *k* steps backward on the currently explored path to continue path exploration.

SAGE [64] implements a generational search for multiple path exploration with a coverage-optimized heuristic to improve coverage. Thummalapenta *et al.* [129] and Ma *et al.* [94] both confirmed that approaches such as PEX, KLEE, and SAGE without the ability to precisely focus on a particular code element may not be sufficient to improve coverage and to enhance error detection capabilities. Baluda *et al.* [12] proposed ARC-B by combining dynamic symbolic execution and abstraction refinement for structural coverage improvements and infeasible code identification. The integration of abstraction refinement can be beneficial by helping to get rid of unreachable code and then forwarding dynamic symbolic execution to explore uncovered but reachable code to discover more structural coverage. On the test subjects experimented, the authors showed that ARC-B outperforms both CREST and KLEE.

The work of Xiao *et al.* [142] introduced cooperative developer testing where software engineers provide guidance to help structural coverage testing tools achieve high structural coverage and the testing tools provide feedbacks to software engineers on the problems identified. This work emphasizes the facts that the reachability problem of code elements is, in general, undecidable and that while tools can help to increase the degree of automation, human intervention is necessary.

Program slicing has been also utilized to perform path exploration for exposing changes in different versions of a program for test suite augmentation in regression testing [113], [123]. However, since both control and data dependencies are taken into account, the resulting slice may be too large to explore. Note that a single small piece of code can yield a number of paths too huge to exhaustively be explored.

The approach developed during the course of this research project therefore employs data dependency analysis to carry out path exploration of dynamic symbolic execution toward effectively and efficiently achieving high structural coverage. When compared to the approaches surveyed above, our proposed approach differs by directing path exploration to focus on data dependencies in order to influence the execution of given code elements. This feature is important as dynamic symbolic execution faces the path explosion problem and the execution of considerable portions of code does not depend directly on the symbolic input. Our proposed approach is presented in detail in the following section.

5.3 Approach

In the context of structural coverage testing, completely achieving the path coverage criterion is practically infeasible. Besides, testing large and complex software programs and referring to sophisticated test adequacy criteria is often beyond the scope of a typical testing budget. In the practice of software development, the requirement of high structural coverage achievements such as statement and branch coverage has been long advocated as a convenient way to measure the adequacy of software testing [20], [56], [117], [132]. Accordingly, the second objective of this research project is to enhance statement and branch coverage adequacy for the program under test. The automated test input generation technique being utilized is dynamic symbolic execution and the ultimate goal is to improve the solving of the reachability problem, which improving statement and branch coverage criteria can essentially be reduced to.

As noted above, the use of dynamic symbolic execution for automation of test input generation suffers from the combinatorial explosion problem of the path space. To cope with this challenging scalability issue, we have developed a goal-oriented testing approach to significantly improve the efficiency of path exploration in dynamic symbolic execution. A test goal is actually a statement or a branch of the program under test. The testing approach makes use of data and control dependencies to carry out path exploration to quickly explore the test goal. The application of this testing approach to obtaining the statement coverage criterion and the branch coverage criterion is straightforward. Figure 5.2 sketches the structural coverage testing algorithm implemented in this study.

Algorithm 6 Structural Coverage Testing

Input : Program P, Testing limit *limit*
Output : Set of test inputs T, Structural coverage result C

```
1:  CFG ≔ ComputeCFG(P)
2:  while termination conditions are not reached do
3:      g ≔ SelectUncoveredElement(CFG)
4:      l ≔ CalculateTestingLimit(limit)
5:      t ≔ Guider(P, g, l)
6:      if t is not null then
7:          T ≔ T ∪ { t }
8:          C ≔ UpdateCoverage(CFG, C, g)
9:      end if
10: end while
11: return (T, C)
```

Figure 5.2: A structural coverage testing algorithm using the goal-oriented dynamic test generation approach

The algorithm takes as input a program under test P and a testing limit *limit*. The output is a set of test inputs T together with the desired structural coverage percentage achieved C. The algorithm is terminated when one of the following termination conditions is satisfied (line 2). The first condition is if the structural coverage criterion being attempted has been completely achieved. The second condition is if the testing limit has been no longer valid.

For each iteration of the **while** loop, the algorithm carries out the following steps. First, it selects an uncovered coverage element g and also calculates a testing limit l for exploring g (lines 3–4). Now, given the program under test P, the coverage element g, and the testing limit l, the algorithm invokes GUIDER to find a test input t so that the execution of the program P on test input t executes g (line 5). GUIDER uses dynamic symbolic execution to perform test input generation, and exploits control and data dependencies to improve the efficiency of path exploration (the detailed description of GUIDER was given in section 4.5). Once GUIDER is invoked, a test input t is returned, and the test set T and the structural coverage result C are updated accordingly (lines 6–9).

To this point, we have presented all the necessarily theoretical elements underlying the structural coverage testing approach developed during the course of this research project. We now proceed to describe the proposed structural coverage testing framework, the implementation, and also the preliminary experiments conducted to

evaluate the effectiveness of our testing approach in comparison with popular state-of-the-art testing approaches.

5.3.1 The Proposed Testing Framework

In this subsection, we describe the architecture of the structural coverage testing framework implementing the testing approach proposed in Figure 5.2. It is named **S**tructural **C**overage **T**esting (SCT). SCT is designed for branch coverage testing and works on programs written in C. The SCT framework is built on top of CREST [19], an automatic test input generator for C based on dynamic symbolic execution. CREST relies on CIL [99] for the instrumentation and static analysis of C code, and on the YICES SMT solver [47] for constraint solving.

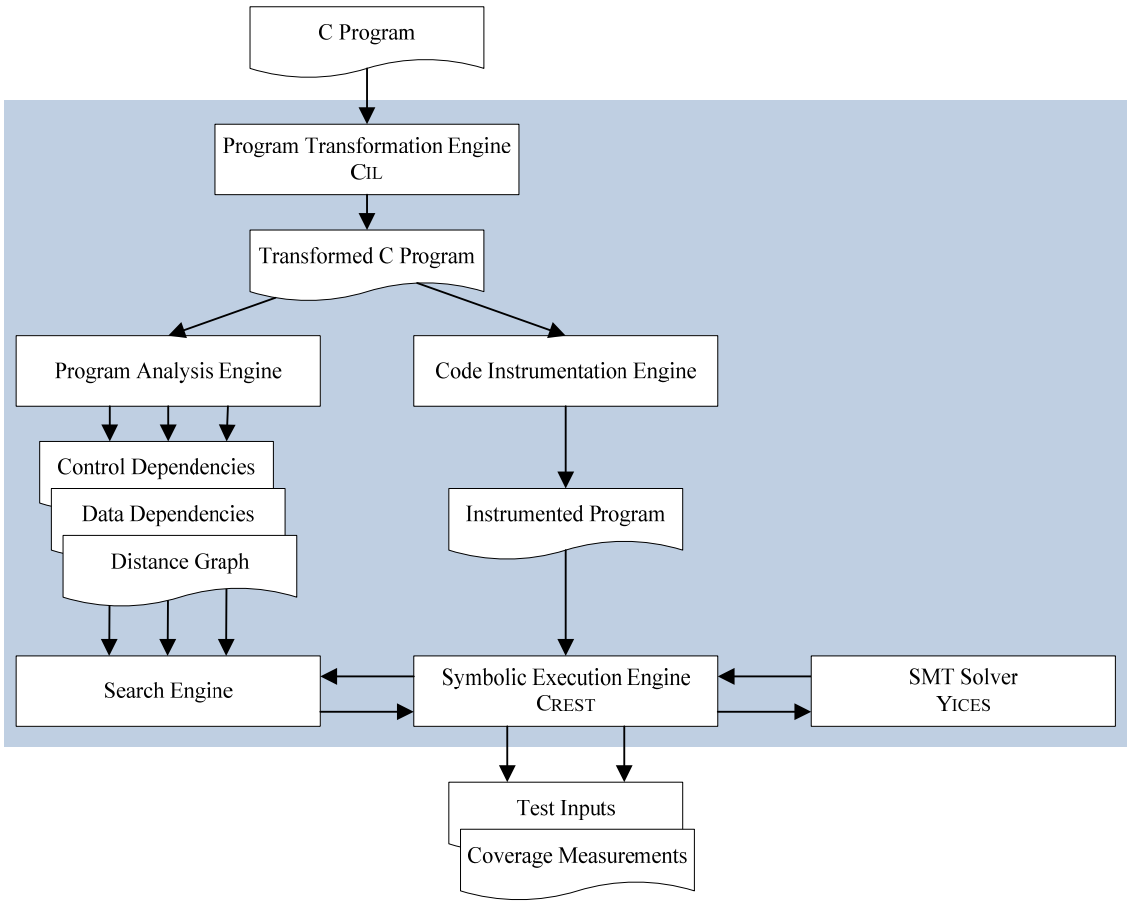

Figure 5.3: The proposed structural coverage testing framework

Figure 5.3 shows the logical components and the basic workflow of the SCT framework, and also illustrates how it extends the functionality of the CREST platform to automate test input generation for structural coverage testing. The straight-lined rectangles indicate the logical components in SCT while the curved rectangles indicate input/output

relationships between these components. The interaction between the logical components is captured by arrows indicating the execution flow as well as both the control and data dependencies.

Given a C program under test P, the SCT framework first uses CIL to parse, type check, and transform P into a simplified C program P', or a subset of C. The main advantage of using CIL is that it simplifies the source code of the program under test P into a few core constructs with a clean semantics to facilitate static program analysis, runtime annotations, and code instrumentation. The CIL front-end enables SCT to easily analyse and manipulate C programs transparently through a syntax-directed type system. In this transformation phase, SCT particularly normalizes the structure of the transformed program P' and then computes its control flow graph model to begin structural coverage testing. A detailed description of the normalization is provided in the following subsection.

In the development of the SCT framework, one of the most important components is the *program analysis engine*. Since SCT implements the goal-oriented testing approach presented in the previous chapter, it computes control dependencies, data dependencies, and shortest paths between statements in the static structure of the program. Control dependencies indicate the relationship of the execution flow between statements while data dependencies indicate data flows affecting the execution of statements. The identification of the shortest paths between statements potentially helps to navigate path exploration to quickly discover a specific path among the large path space of the program, leading to the execution of a desired code element. In this logical component, SCT therefore extends CIL to implement control dependency analysis, data dependency analysis, and also compute minimal branch distances. The result of these analyses is used by the *search engine* to effectively and efficiently perform path exploration in dynamic symbolic execution for structural coverage testing.

To be able to carry out dynamic symbolic execution, the source code of the transformed program is instrumented through the *code instrumentation engine*. The instrumentation phase captures and simulates the semantics of every statement being executed at runtime. For this, it inserts additional code before each statement to enable tracking memory addresses and runtime values of the variables involved in the statement, and also to allow operators to be executed concretely and symbolically. Therefore, this

phase enables the program under test to be simultaneously executed both concretely and symbolically.

In the SCT framework, the structural coverage testing process is operated through the communication between the following three logical components, *Search Engine*, *Dynamic Symbolic Execution Engine*, and *SMT Solver*. The *Search Engine* component determines how the path space of the program is to be explored for structural coverage testing. It relies on the current coverage achievement as well as the result of dynamic symbolic execution to drive path exploration. The core part of *Search Engine* is the GUIDER search algorithm proposed in chapter 4. *Dynamic Symbolic Execution Engine* performs dynamic symbolic execution through the guidance from *Search Engine* and the interaction with the underlying *SMT Solver* (YICES) for test input generation. It also tracks the execution of statements and branches in the program for coverage measurements.

5.3.2 Implementation

As noted above, we implemented the SCT structural coverage testing framework on top of the CREST platform. SCT extends CIL with approximately 8000 lines of OCaml code to implement the logical component *Program Analysis Engine*. For *Search Engine*, SCT implements this logical component as a search heuristic in CREST with about 4000 lines of C/C++ code.

The structural coverage criterion that SCT attempts to address is branch coverage. To best achieve this structural coverage criterion, SCT, based on CREST, unrolls decisions with multiple conditions as an equivalent cascade of single condition decisions and converts every single conditional statement into the form of *if (e) then* S_1 *else* S_2. The branch coverage criterion achieved by SCT is therefore comparable to condition coverage in the original program. For any specification clothed in forms of assertion calls, i.e. *assert (e)*, SCT transforms it into a conditional statement *if (!e) error()* to check for violations. Table 5.1 provides examples to illustrate these transformation details in SCT.

The SCT framework does not automatically identify the input interface of the program; rather users must determine and annotate input variables to form the test driver for the program under test. An example of this activity is given in Table 5.2. This is the only step where human intervention is required when using the SCT framework.

Table 5.1: Examples of converting code snippets into simplified constructs in the SCT framework

Original Code	Transformed Code

`if (r == 0) {`
`  error = true;`
`}`
	`if (r == 0) {`
	`  error = true;`
	`} else {`
	`  // Empty block`
	`}`

`if (r1 == 0 && r2 == 0) {`
`  error = true;`
`}`
	`if (r1 == 0) {`
	`  if (r2 == 0) {`
	`  error = true;`
	`  } else {`
	`  // Empty block`
	`  }`
	`} else {`
	`  // Empty block`
	`}`

`if (r1 == 0 || r2 == 0) {`
`  error = true;`
`}`
	`if (r1 == 0) {`
	`  error = true;`
	`} else {`
	`  if (r2 == 0) {`
	`  error = true;`
	`  } else {`
	`  // Empty block`
	`  }`
	`}`

`assert(e);`	`if (e == 0) {`
	`  error();`
	`}`

Table 5.2: An example of creating a test driver for the program under test in SCT

Original Code	Test Driver

`bool func (int r1, int r2) {`
`  // Function body`
`}`
	`bool func (int r1, int r2) {`
	`  CREST_int(r1);`
	`  CREST_int(r2);`
	`  // Function body`
	`}`

5.4 Evaluation

The primary objective of this research project is to explore automated techniques to improve the efficiency of software testing. In this chapter, we have presented the SCT framework, our proposed approach for automated structural coverage testing for C programs. The development of SCT is based on dynamic symbolic execution to perform

test input generation and utilizes the chaining guided search algorithm GUIDER to carry out path exploration toward effectively and efficiently exploring coverage elements in the program under test. The structural coverage criterion to be addressed by the SCT framework is branch coverage. In this section, we present an evaluation of how effective the proposed approach is in supporting branch coverage testing. In particular, we evaluate the effectiveness of the SCT by considering the following two research questions:

- How does SCT perform in comparison with popular testing approaches in relation to branch coverage improvements?
- How does SCT optimize testing effort in comparison with popular testing approaches in relation to branch coverage improvements?

The first research question focuses on assessing the capability of the SCT framework in improving branch coverage testing. The second question measures how effectively SCT can cope with the path space explosion suffered by dynamic symbolic execution to accomplish testing with high branch coverage achievements. In the following subsections, we present (1) the set of selected test subjects, (2) our evaluation methodology, (3) the experimental results, and (4) the discussions.

5.4.1 Test Subjects

An overview of the test subjects selected to carry out the experiments in this evaluation is given in Table 5.3. They include the *sample*, *testloop*, and *hello_world* functions. The first function was adopted from the work of Ferguson and Korel [52] while the last two functions were adopted from the work of Xie *et al.* [141]. The next six test subjects, *valves_nest<i>* with *i = {2, 5, 10, 20, 50, 100}*, are from the work of Baluda *et al.* [12]. These test subjects are employed in the literature to illustrate the essence of individual exploration problems of dynamic symbolic execution. Thus we wanted to check if SCT, by exploiting control and data dependencies, was able to tackle these exploration problems. In particular, for the *valves_nest<i>* programs where, as reported in [12], *depth-first search* such as DART [62] and CUTE [124], *control-flow graph directed search* in CREST [19], and KLEE [26] all exhibited their worst performance and scored very low structural coverage, we expected that SCT, with the ability to precisely identify the root cause of the execution of hard-to-reach code elements, could potentially guide path exploration to achieve maximal branch coverage within minimal exploration effort.

Notice that the *hello_world* function in this evaluation was modified to check an input array which must start with "Hello", end with "World!" and contain only spaces. This modification makes the function more difficult for search strategies to cover all its branch coverage.

The rest of the test subjects were mentioned in the work of Binkley *et al.* [16]. These functions come from open-source programs and we hence wanted to evaluate the capability of SCT in dealing with the high complexity of real world programs. For the sake of these experiments, for some functions we just extracted part of their code.

All the test subjects were in C code and to enable comparison with tools such as PEX [128] and FITNEX [141], we converted them to C# code.

Table 5.3: An overview of the test subjects selected in the evaluation of the SCT framework for structural coverage testing

Subject	# Loc	# Statements	# Branches
sample	31	48	12
testloop	17	27	8
hello_world	37	92	32
valves_nest2	32	60	12
valves_nest5	62	132	30
valves_nest10	112	252	60
valves_nest20	212	492	120
valves_nest50	512	1212	300
valves_nest100	1012	2412	600
netflow	28	28	6
moveBiggestInFront	37	32	6
handle_new_jobs	37	25	6
update_shps	52	47	10
check_ISBN	78	218	52
check_ISSN	78	210	52
Total	**2337**	**5287**	**1306**

5.4.2 Methodology

To evaluate the effectiveness of our proposed approach, we conducted experiments on the selected test subjects, and compared SCT with two widely adopted search strategies, *random input search* (or RANDOM) and *depth-first search* (or DFS), and with three

automated test input generation tools: CREST [19], PEX [128], and FITNEX [141]. For CREST, we chose the *control-flow graph directed search strategy* (or CFGDIRECTED), which was confirmed as the "best" search algorithm by the experimental data. PEX implements dynamic symbolic execution to generate test inputs for .NET code, supporting languages C#, VisualBasic, and F#; FITNEX is an extension of PEX.

All experiments in the evaluation were run on 3GHz CoreTM2 Duo CPU with 4GB of RAM, running Ubuntu GNU/Linux with a 32-bit kernel version 3.2.0–38 for RANDOM, DFS, CFGDIRECTED, and SCT, and running Windows 7 for PEX and FITNEX.

As our primary purpose was to evaluate the capability of each tool (or search strategy) in achieving high branch coverage, it was fair to set up a fixed testing budget for all. For this, time can be an option. An alternative option can be the number of explorations, or runs, to explore the path space of the program under test. In the context of applying dynamic symbolic execution where the underlying search strategy must optimize the path space explosion to maximize the achievement of the testing goals, the second option is preferable and is commonly adopted in testing tools such as CREST, PEX and FITNEX. In this particular evaluation, we therefore adopted the second option. Considering the relatively small-sized test subjects, we specifically chose 1000 runs as the testing limit to run every test subject on each tool. We measured the percentage of branch coverage obtained and the results are shown in Table 5.4.

Our secondary purpose was to evaluate the capability of each tool in optimizing path exploration for branch coverage improvements. This is an important criterion for evaluating the effectiveness of any test input generation tool based on dynamic symbolic execution because the cost of performing dynamic symbolic execution is expensive, and minimizing the number of path explorations is necessary to improve the applicability of the technique. For this, besides the first stop condition (1000 runs), we also stopped the tools when the branch coverage criterion of the experimenting test subject had been completely achieved. The results are given in Table 5.5.

5.4.3 Experimental Results

Table 5.4 and Table 5.5 summarize the statistics obtained from the experiments. It is clear from the statistics that RANDOM is the worst approach to test input generation, with the lowest average coverage (40%) obtained but the highest average number of runs (891 runs) exploited. Remarkably, on the test subjects *valves_nest<i>*, RANDOM

Table 5.4: Percentage of branch coverage achieved by search strategies on 15 test subjects

Subject	RANDOM	DFS	CFGDIRECTED	PEX	FITNEX	SCT
sample	42	92	92	92	92	100
testloop	13	88	88	100	100	100
hello_world	34	56	91	91	91	100
valves_nest2	**17**	**42**	**42**	100	100	100
valves_nest5	**7**	**17**	**17**	100	100	100
valves_nest10	**3**	**8**	**8**	100	100	100
valves_nest20	**2**	**4**	**4**	80	100	100
valves_nest50	**1**	**2**	**2**	**39**	**58**	100
valves_nest100	**0**	**1**	**1**	**20**	**25**	100
netflow	83	83	83	100	100	100
moveBiggestInFront	100	83	83	100	100	100
handle_new_jobs	67	100	100	83	100	100
update_shps	60	90	90	100	100	100
check_ISBN	83	83	83	96	83	98
check_ISSN	83	83	94	96	83	98
Average	**40**	**56**	**59**	**87**	**89**	**100**$^{\approx}$

Table 5.5: Measurements of numbers of program explorations performed by search strategies

Subject	RANDOM	DFS	CFGDIRECTED	PEX	FITNEX	SCT
sample	1000	1000	1000	1000	1000	13
testloop	1000	1000	1000	26	27	22
hello_world	1000	1000	1000	1000	1000	55
valves_nest2	_	1000	1000	92	71	11
valves_nest5	_	1000	1000	236	125	26
valves_nest10	_	1000	1000	338	741	51
valves_nest20	_	1000	1000	1000	689	101
valves_nest50	_	1000	1000	1000	1000	251
valves_nest100	_	1000	1000	1000	1000	501
netflow	1000	2	1000	5	5	2
moveBiggestInFront	15	1000	1000	4	4	2
handle_new_jobs	1000	2	2	1000	99	34
update_shps	1000	1000	1000	5	7	4
check_ISBN	1000	1000	1000	**234**	**313**	51
check_ISSN	1000	1000	1000	**234**	**313**	45
Average	**891**	**867**	**934**	**478**	**426**	**78**

with its *careless* input generation mechanism ran out of memory due to becoming trapped in too-long loop iterations requesting large amounts of memory. We stopped RANDOM after 25 runs for these test subjects and recorded the coverage achievement. We did not accumulate these numbers of runs into its average run number since it does not make sense to make the comparison. These test subjects appear to be the biggest hurdles for current test input generation tools. For instance, RANDOM could not reach beyond 2 branches on any of these test subjects.

DFS is an instance of using dynamic symbolic execution to systematically explore all feasible paths of a program. It lacks the ability to forward the execution to further unexplored control flow points and hence achieved very low branch coverage within the fixed testing limit. However, since DFS relies on the power of the underlying constraint solver, it obtains higher coverage (56% on average) than RANDOM.

With CFGDIRECTED, 14 out of 15 cases failed to achieve full coverage. For these cases, the test subjects contained branches that required precise guidance of data flow analysis to be covered. CFGDIRECTED only utilizes the static control flow graph and thus is not effective. CFGDIRECTED achieved coverage only slightly higher than DFS (59% on average). However, on the test subjects *valves_nest<i>*, DFS and CFGDIRECTED both exhibited their worst capability in coping with the combinatorial explosion of the path space to achieve high coverage.

PEX and FITNEX achieved quite similar average coverage results: 87% and 89%, respectively. While PEX failed in 8 cases to achieve full coverage, FITNEX failed in 6. In cases of *check_ISBN* and *check_ISSN*, both PEX and FITNEX automatically terminated after 234 and 313 runs, respectively, although total coverage was not achieved. The comparison thus favours these tools with respect to path explorations. The results obtained by both PEX and FITNEX are better than RANDOM, DFS, and CFGDIRECTED in terms of coverage achievements and exploration optimizations. This highlights the power of bringing several search strategies together as well as the power of fitness functions in test input generation. However, on the test subjects *valves_nest50* and *valves_nest100*, both the approaches failed to expose high complexity code.

SCT failed to achieve 100% coverage in 2 cases, *check_ISBN* and *check_ISSN*. We manually investigated these test subjects and realized that the two functions contained one unreachable branch that resulted from the instrumentation step where our tool

normalizes every *if* statement to have the form *if (e) then S1 else S2*. Currently, SCT is not able to deal with infeasible code. But an interesting observation when we conducted experiments on these test subjects was that even though we set the testing limit to 1000 runs, SCT stopped the exploration process after 51 runs for *check_ISBN* and 45 runs for *check_ISSN*. This means the search process considered all possible combinations of data flows but none could help to explore the test goal. This suggests that this code element was infeasible. We refer this situation to *saturated data propagation* and are working on a formal proof for identifying infeasible code by exploring data flows.

Noticeably, with the six test subjects *valves_nest<i>*, while Random ran out of memory, DFS and CFGDIRECTED both revealed problematic issues in their search mechanism, PEX failed to reach even 50% for *valves_nest50* and *valves_nest100*, and FITNEX also succumbed to these two test subjects due to overly long execution traces. SCT not only scored 100% coverage but also efficiently minimized the exploration process, using less than one run to explore one branch on average.

It is significant that on average SCT achieved the highest coverage (100% if infeasible code is not counted) and maintained a significantly small number of program explorations (78 runs compared to 426 and 478 for FITNEX and PEX, respectively, and 934 for CFGDIRECTED) on the selected test subjects. This shows the capability of utilizing data flow analysis to guide dynamic symbolic execution in the test input generation process.

In this evaluation, we did not conduct experiments to compare our proposed approach SCT with the well-known dynamic symbolic execution KLEE tool [26] or with the ARC-B tool [12]. The latter was proposed to combine dynamic symbolic execution and abstraction refinement to mitigate the combinatorial explosion of the path space for coverage improvements and infeasible code identification. However, the experimental data reported in Baluda *et al.* [12], specifically on the sequence of test subjects *valves_nest<i>*, reveals that the KLEE tool, like DFS and CFGDIRECTED, obtained very low coverage, covering only 5 branches in spite of the increasing size of the test subjects. In case of *valves_nest100*, DFS, CFGDIRECTED and KLEE did not go beyond 1% coverage.

The ARC-B tool, when compared to SCT again on the test subjects *valves_nest<i>*, demanded considerable numbers of program explorations to acquire the same coverage

as SCT did, almost 6 times as many. Apart from that, on the test subject *valves_nest100*, ARC-B reached 71% coverage after 10,000 runs. SCT not only scored 100% coverage but also successfully accomplished the search process after 501 runs, or approximately 20 times fewer.

5.4.4 Discussion

A preliminary evaluation to evaluate the effectiveness of the SCT framework for structural coverage testing was carried out on 15 selected test subjects selected. The primary purpose was to evaluate the ability of SCT to improve branch coverage achievements. The secondary purpose was to assess the ability of SCT to optimize path exploration to maximize coverage testing results within the constraint of testing limits. The evaluation measured these two aspects by comparing the testing results achieved by SCT with those achieved by state-of-the-art structural coverage testing tools such as RANDOM, DFS, CREST, PEX and FITNEX. The experimental results highlight the efficiency of exploiting program dependencies such as control and especially data dependencies to perform dynamic symbolic execution toward effectively and efficiently exploring code elements of the program under test.

In the application of SCT to structural coverage testing, the cost of running SCT comes mostly from two areas. One is the cost of computing control and data dependencies, and a distance graph. These computation costs are minor compared to the very expensive cost of carrying out the dynamic analysis technique dynamic symbolic execution. Take the case of computing data dependencies to identify definition statements, for example. It involves a maximal fixed-point algorithm operated statically on the source code of the program under test prior to dynamic symbolic execution. The algorithm complexity is the product of the height of the lattice and the number of nodes in the control flow graph. The cost of performing dynamic symbolic execution with the guidance of event sequences depends on the number of runs that SCT requires to execute the program under test, which was found to be significantly smaller than other search algorithms and tools. In fact, we observed from the experiments that CFGDIRECTED and SCT both executed the test subjects within a matter of a few seconds. PEX and FITNEX, however, consumed a considerable amount of time on all the test subjects.

In this evaluation, we conducted the experiments on a collection of relatively small test subjects. These test subjects, however, have been already used in the software testing

literature to demonstrate problematic exploration issues of recently proposed techniques using dynamic symbolic execution. The following chapter sets out to verify the validity of this evaluation and we discuss there the limitations of the current implementation of our proposed goal-oriented approach for both structural coverage testing and security vulnerability detection. Nevertheless, we believe that when testing sizeable and complex programs, where the path space of the program under test is exceedingly huge to be systematically and exhaustively explored, the ability of our proposed approach to break down the path space and to precisely guide the path exploration process by focusing on selected aspects of semantics is essential for optimizing the expensive cost of performing dynamic symbolic execution to maximize structural coverage achievements and to enhance error-detection capabilities.

5.5 Summary

The state space of software programs is generally infinite; exhaustively exploring and checking it is beyond current computational power. In the practice of software development, software testing is a widely adopted method for improving software quality and reliability. It executes the program under test and allows software engineers to observe actual behaviours during the program execution. Testing and checking all possible combinations of the input values is potentially endless. In the face of this reality, researchers have developed test adequacy criteria to give powerful measurements of how adequately software testing is carried out in accordance with software quality factors.

In this chapter, we focused on structural coverage testing, a relatively simple set of test adequacy criteria that has been widely adopted in software development practice to support software quality and reliability [20], [56], [117], [132]. We proposed the SCT framework to automate structural coverage testing for achieving the branch coverage criterion. The development of SCT utilized the chaining guided search algorithm GUIDER proposed in chapter 4 to effectively and efficiently improve the efficiency of path exploration under dynamic symbolic execution for high branch coverage achievements. The experimental results show that SCT is effective in maximizing structural coverage, optimizing path exploration, and providing useful evidence to identify infeasible code elements. In most of the experiments, SCT was able to achieve

higher branch coverage with significantly fewer path explorations than popular state-of-the-art test input generation approaches.

This chapter addressed the second objective of the research project during this course: exploring automated techniques to effectively and efficiently improve structural coverage testing. Section 5.1 provided the basic background for structural coverage criteria. Section 5.2 surveyed automated test input generation techniques for structural coverage testing. Section 5.3 described the development of the SCT framework and its implementation details. In section 5.4, preliminary experiments were conducted to evaluate the effectiveness of SCT by measuring its ability to improve branch coverage results as well as its ability to optimize path exploration in comparison with existing approaches. The experimental results show the efficiency of SCT in utilizing data dependency analysis to deal with the path explosion problem facing dynamic symbolic execution for solving the reachability problem.

Chapter 6

Security Vulnerability Detection

In the previous chapter, we presented SCT, our proposed framework for structural coverage testing. SCT employs the chaining guided search algorithm GUIDER to perform dynamic symbolic execution for improving branch coverage results. The observation was that having achieved high structural coverage of the program under test may give confidence that the program has been thoroughly tested and less likely to harbour errors. For example, in the adoption of the statement coverage criterion, every statement must be exercised and checked at least once. But the state space of a program is potentially endless and one statement at runtime can encode an infinite number of states. As such, only one state is verified for desired software quality factors and the adoption of the statement coverage criterion is not sufficient for ensuring software quality and reliability. Furthermore, among the many states not being exercised and verified, there can be buggy or *even unsafe* states that can be exploited to cause serious software failures such as security breaches, damaging the data and functionality of the host system. In this chapter, we focus on software security.

With the development of the Internet, the application of software systems has suffered seriously from security problems. Security vulnerabilities are discovered daily in commonly used software [37], [101]. Approximately half of all security vulnerabilities

are detected at the source code level [102]. Such vulnerabilities can be exploited to devastate the data and functionality of the host system, leading to significant financial losses [63], [122]. Testing to identify and remove security vulnerabilities, therefore, has been becoming essential in the current software development practice.

There are various types of security vulnerabilities [48] and in this research we particularly focus on buffer overflow vulnerabilities, which represent one of the most serious classes that security threats [41]. A buffer overflow is an anomaly where a program, while assessing a buffer, overruns the buffer boundary. This can be exploited to open security breaches, causing erratic program behaviours. In this chapter, we present an approach to detect such buffer overflow defects by combining static runtime verification and dynamic symbolic execution. The chapter is structured as follows. In section 6.1, we describe buffer overflow vulnerabilities and how they can be exploited to open security breaches. Section 6.2 conducts a literature review to survey techniques proposed to detect and eliminate buffer overflow detects. Section 6.3 presents our proposed approach for buffer overflow detection. The approach works in two phases. In the first phase, it uses DEPUTY [31] — a novel type system for pointers — to diagnose potential runtime violations on buffer operations in the program under consideration. In the second phase, it uses dynamic symbolic execution to generate test inputs to uncover violations. In order to effectively and efficiently perform dynamic symbolic execution, we employ the chaining guided search algorithm GUIDER proposed in chapter 4 to guide path exploration for test input generation. Section 6.4 elaborates an evaluation conducted to assess the effectiveness of our proposed approach against 23 buffer overflow vulnerabilities compared with popular state-of-the-art search algorithms. Finally, a summary of the chapter is given in section 6.5.

6.1 Overview

A security vulnerability is a weakness allowing an attacker to compromise the availability, confidentiality or integrity of a computer system. Security vulnerabilities may be the result of a programming error that yields security holes such as information leakage, modification, and destruction. An attack is a successful exploitation of security vulnerabilities. There are various types of security vulnerabilities occurring at the source code level, such as buffer overflows, format string bugs, SQL injections, cross site scripting, and cross site request forgery [40]. We mainly focus here on buffer overflow

vulnerabilities in C programs, which have been widespread in both legacy and modern systems and account for nearly half of all known security vulnerabilities [135]. Buffer overflows are one of the top 25 most dangerous programming errors [41]. They affect the security of programs written in languages that are neither type- nor memory-safe such as C/C++, enabling an attacker to gain control of the host computer and execute injected code with elevated privileges. In this section, we illustrate the attack mechanism through exploiting buffer overflow vulnerabilities.

A buffer overflow vulnerability occurs when data can be written outside the memory allocated for a buffer, overwriting the content of the neighbouring memory segments [103]. The presence of buffer overflows might be due to library function calls, pointers, aliases, logic errors, and a lack of bound checks when accessing buffers in low level programming languages such as C, C++, and x86 assembly. One of the most common buffer overflows is due to erroneous pointer arithmetic operations on buffers. For instance, `p = q + index` results in a pointer `p` pointing to a location `index` bytes away from the current location of pointer `q`. Now, if a write operation is performed through dereferencing pointer `p`, i.e. `*p = '0'`, and the location of `p` is outside the allocated memory for the buffer, then this write operation can overwrite the data being stored at `q + index`.

In a buffer overflow attack, the goal is to spoil the functionality and then take control of a privileged program. If the program being attacked is sufficiently privileged, the attacker gains control of the host system to execute the injected code. To successfully accomplish this goal, the attacker must achieve the following steps:

- Exploit weaknesses of a software program to trigger a buffer overflow attack.
- Arrange suitable code into the address space of the program under attack.
- Navigate the execution flow to jump to the injected code with suitable privileges and parameters loaded into registers and memory.

The ability to explore buffer overflows depends on the memory regions allocated to buffers such as stack, heap, data segments, and bss segments. In the remainder of this section, we mainly discuss stack-based buffer overflow attacks.

```
(s)  void overflow (char * str) {
(1)      int i;
(2)      char buf[32];

(3)      for (i = 0; i < strlen(str); i++) {
(4)          buf[i] = str[i];
(5)      }

(6)      return;
(e)  }
```

Figure 6.1: A C program to illustrate the stack-based buffer overflow attack

In a stack-based buffer overflow attack, the memory allocation of the buffer resides in the stack memory region. Consider the C program `overflow` in Figure 6.1 for example. A buffer `buf` is declared at line 2 and is allocated 32 bytes in the stack memory region. The valid location of this buffer is between `buf[0]` and `buf[31]`. The *for* loop in lines 3–5 copies the content of the input buffer `str` into `buf`. A buffer overflow can happen at line 4 if the length of the input buffer `str` is over 32. If so, the loop does allow copying more than the capability that the buffer `buf` can store. To understand the effect of a buffer overflow, consider the stack layout of the program when invoking the function `overflow` given in Figure 6.2. Here, we suppose that `overflow` is invoked from the `main` function and the growth direction of the stack is downward, meaning the memory region allocated when a function is called is in the lower part of the stack.

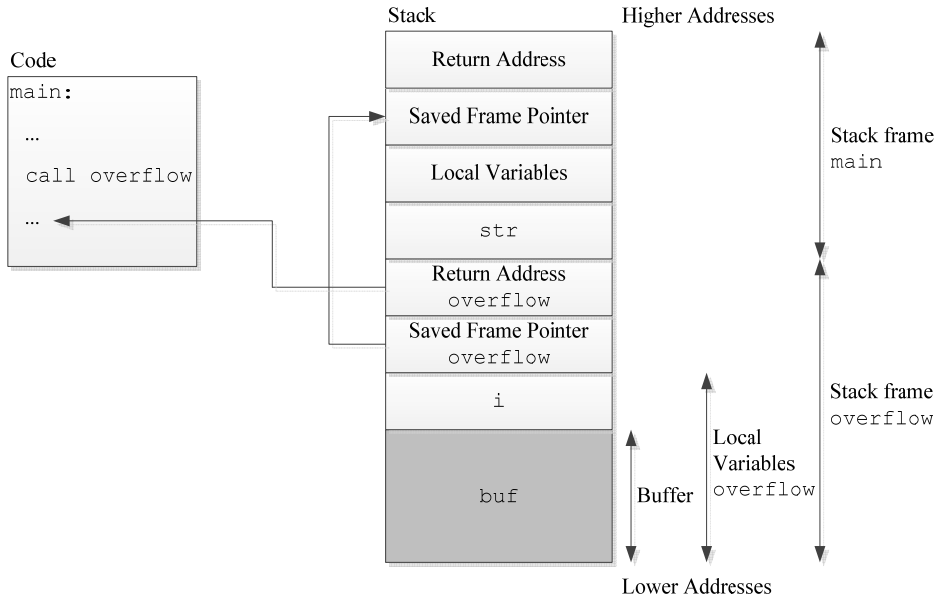

Figure 6.2: The stack layout of the program when calling the `overflow` function

The stack is partitioned into stack frames. Each stack frame stores information about a function being executed, including registers, the return address, the saved frame pointer, and locally declared variables. The return address points to the address of the statement to continue the program execution right after the function called has finished the execution. Any array declared locally is located in the section of local variables of the function stack frame. Since in languages like C there is no information about the array size available at runtime, most C compilers allow copying data going beyond the end of an array. In this case, the management information of the function such as the return address and the saved frame pointer is subverted.

In this example, we assume that the return address and the saved frame pointer of the `overflow` function occupy 4 bytes, whereas `i` variable occupies 2 bytes. The total memory region allocated to execute `overflow` is 42 bytes. The gray coloured part in Figure 6.2 indicates what can be written to by the function if the buffer `buf` is used correctly. Now consider what can happen if an attacker is able to trigger a buffer overflow attack to the `overflow` function. The attack target is to corrupt the return address and execute the injected code through buffer (see Figure 6.3). By manipulating the program to invoke the `overflow` function with 42 bytes as the size of the `str` input parameter, the attacker is able to copy 32 bytes into the `buf` buffer, overwrite variable `i` (2 bytes), the saved frame pointer (4 bytes), and the return address (4 bytes). In order to take control of the program, the attacker changes the return address to point to the

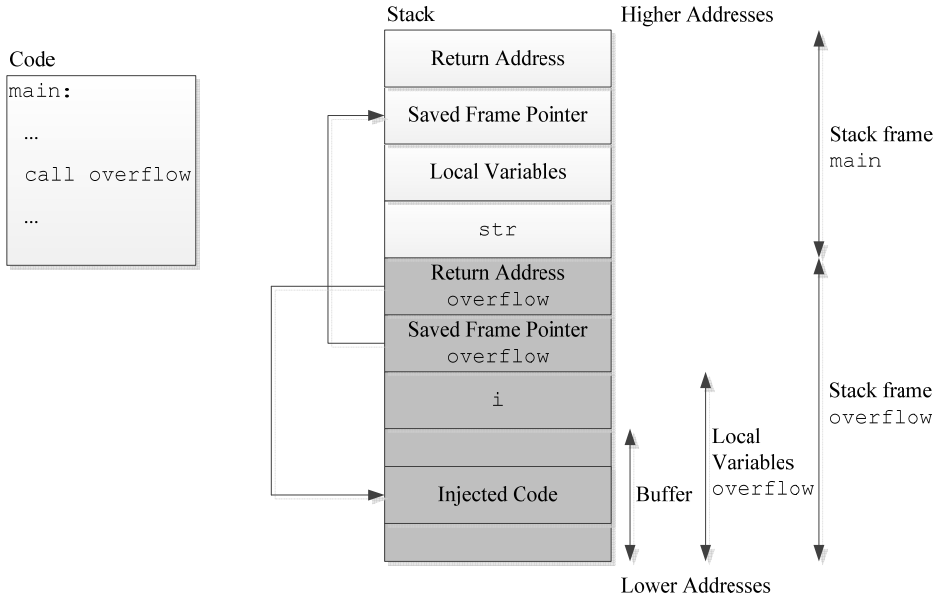

Figure 6.3: Basic stack-based buffer overflow attack

injected code that he already copied into the buffer. Now, when the `overflow` function returns, the return address is usually used to resume the program execution after the function call has finished. However, since the return address has been changed to point to the injected code, the execution flow will continue to execute the attacker's code. The injected code can be a shell code, e.g. "`/bin/sh`", to launch a remote shell with the root privilege. This is referred to as a "return address clobbering" or "direct code injection" attack [49].

In some situations, the attacker might not be able to modify the return address of the called function since the program has enforced some mechanism to protect the return address by some countermeasure. The attacker may still discover an alternative way to execute the injected code by modifying the saved frame pointer [79]. As shown in Figure 6.4, the attacker can modify the saved frame pointer of the `overflow` function to point to a desired location of the buffer. As a result, the first 4 bytes of the buffer are regarded as the saved frame pointer of and the next 4 bytes the return address of the `main` function, the calling function of `overflow`. The content of the modified return address of `main` is changed to point the injected code of the attacker. Consequently, the `overflow` function returns to its calling function as usual but when the `main` returns,

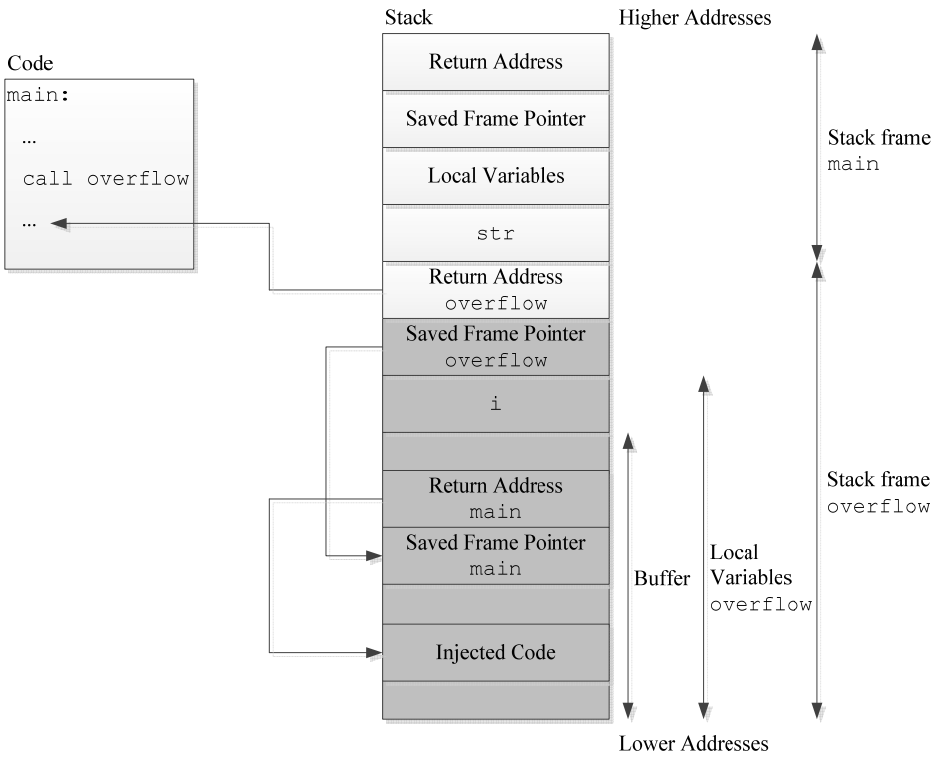

Figure 6.4: Stack-based buffer overflow through overwriting saved frame pointer

the injected code being stored in the stack is activated.

It might be the situation that the attacker is not able to overwrite the saved frame pointer due to the presence of countermeasures that prevent code injections of buffer overflow attacks by converting stack segments from executable to non-executable [106]. The attacker can still bypass such defences, however. This is carried out by exploiting weaknesses in system library functions, for example in "return-to-libc" or "jump-to-libc" attacks [49].

Attacks via buffer overflows happen in programs written in low level programming languages such as C, C++, and x86 assembly, which are not type- and memory-safe. The severity of buffer overflow attacks has been ranked high among software security vulnerabilities since an attack may let the attacker to gain control of the host system and then execute the injected code. Unsurprisingly, buffer overflow vulnerabilities dominate in the area of remote network penetration vulnerabilities [42]. The reality is that there are millions of lines of code invested in existing operating systems and security-sensitive applications, and the vast majority of that code is written in C. For instance, the Java Virtual Machine (JVM) is also a C program, and one of the ways to attack a JVM is to conduct buffer overflow attacks to the JVM itself [44]. Therefore, much research effort has attempted to investigate and develop techniques to mitigate buffer overflow vulnerabilities. In the next section, we provide a brief survey of these buffer overflow vulnerability mitigation techniques to clarify the context of our study.

6.2 Literature Review

Security is an essential concern in the practice of software development, especially as software applications increasingly shift to the Internet. As the accessibility of the software is expanded, the probability that the software is vulnerable to security attacks significantly increases. In the attempt to validate the software against security vulnerabilities, security testing is one of the most proactive techniques being adopted in software industry. The intent is to locate and identify conditions under which the software might open security breaches to attackers [137]. Note that the focus of security testing is not as the same as that of traditional functional testing. In functional testing, testing is performed on behalf of a legitimate user of the software application who is attempting to use it in the way it was intended to be used and for its intended purpose.

Testing from this point of view can cause testers to bypass a large percentage of security tests. This distinction is clarified in the work of Whittaker and Thompson [137], where 19 specific security attacks are presented and many traditional testing techniques used to test software functionality are shown to be unable to discover the security vulnerabilities. In fact, in security testing, testers must behave like attackers, attempting to develop misuse cases, identify assumptions and develop test cases to violate them, identify configuration issues and design test cases to check them, and develop invalid test inputs [4]. In testing to uncover buffer overflow vulnerabilities, for example, every time a software system receives data from its environment, testers must validate if the data violate assumptions of its size.

Security testing based on human intervention such as code review and manual testing is labour-intensive and error-prone. Many security vulnerabilities are subtle and almost invisible to testers [137]. Not surprisingly, commonly used and even well-tested software is exposed to security breaches daily [37], [101]. One attempt to mitigate the hardship of human intervention in security testing utilizes static analysis to automate security vulnerability checking. Over time, several static analysis techniques have been proposed and developed to achieve this goal, including symbolic execution, abstraction interpretation, model checking, integer range analysis, interprocedural analysis, and type inference analysis [32]. Static analysis techniques leverage the fact that programming rules often map clearly to the source code level, thus static inspection can find many of their violations [11]. Considerable research effort has attempted to develop static analysis tools and check security vulnerabilities in real world software applications [31], [32], [42], [80], [135]. Empirical studies have been also carried out to evaluate proposed static analysis techniques and tools [42], [80].

Scalability is perhaps one of the most important advantages of using static analysis techniques. For instance, mature static analysis tools can check millions of lines of code within a matter of a few days and uncover many hidden security defects [11]. Low precision is, however, one of the most challenging issues facing static analysis techniques. It results in false positives, or negative decisions in the presence of software defects, causing users to have low trust in the capability of the tool or even ignore it [11]. Buffer overflow vulnerabilities appear to be even harder for static analysis tools to detect since they often involve complex pointer arithmetic operations on buffers passing through various function calls.

In the attempt to alleviate the low precision of static analysis techniques, and therefore to be able to uncover even more and deeper security vulnerabilities, dynamic analysis has been long advocated in the practice of software testing to carry out security testing [17], [68]. Fuzz testing is a dynamic analysis technique, also referred to as *black-box random* testing, which randomly mutates well-formed inputs, executes the program under test, and verifies it on the resulting data [91]. Fuzz testing is a preferred security testing technique in software industry, thanks to the ability to effectively and efficiently find serious security vulnerabilities [54], [61], [82], [90]. An important limitation of fuzz testing, however, is that the probability of generating particular test inputs to trigger buggy behaviours or to explore corner case branches can be astronomically small. It is well understood that random testing usually provides low code coverage [104]. This drawback implies that potential security vulnerabilities such as buffer overflows can be missed since the code containing the vulnerabilities is not even exercised.

To intertwine the strengths of static analysis and dynamic analysis, dynamic symbolic execution has been proposed with the intent of offering a *directed automated* search mechanism for test input generation [62]. The technique is then further developed in the form of *active property checking* to significantly strengthen automated security testing [65]. It incorporates random testing, symbolic execution, constraint solving, and runtime property checking to *actively search* for property violations such as buffer overflows. In active property checking, it injects at runtime additional symbolic constraints that, when solvable by the underlying constraint solver, will generate new test inputs leading to property violations. To understand active property checking, consider the `divide` example given in Figure 6.5. It takes as input two integer variables, `n` and `d`, and computes their division. A division-by-zero error will occur at line 2 if the value of the denominator `d` is equal zero. To catch this error, traditional runtime checking tools such as Purify [72] and Valgrind [100] would simply check if the value of `d` satisfies `(d == 0)` before the execution of the division operation. The

```
(s)  void divide (int n, int d) {
(1)       // division-by-zero error if d is equal zero
(2)       return (n / d);
(e)  }
```

Figure 6.5: A C program that enables active property checking

check is performed given a specific program execution. In other words, this particular approach is not able to infer which values of `d` can cause the division-by-zero error, or violate given program properties. Testing this program using random testing is unlikely to detect the error since `d` has only one chance out of 2^{32} to be zero if `d` is a 32-bit integer. Dynamic symbolic execution with the attempt to explore all the feasible paths of the program is also likely to miss the error, because the program has only a single path which can be covered no matter what inputs are used. However, if the division-by-zero check `if (d == 0) error();` is inserted into the `divide` program right before the division operation, dynamic symbolic execution can generate an input with `d` equal zero to reveal the error. This observation is the main idea behind active property checking.

Active property checking received a lot of attention from the research community recently. It has been implemented in several tools and used to uncover a number of serious security vulnerabilities of real world software systems [29]. For example, SAGE is a white-box fuzz testing tool developed at Microsoft Research [66]. It implements active property checking to detect a wide range of security errors. A particular focus of SAGE, however, is integer bugs, such as overflow / underflow, width conversions, and signed / unsigned conversions errors. In this research project, we implemented active property checking to check buffer overflow vulnerabilities. One of our research objectives is to improve the efficiency of path exploration in dynamic symbolic execution to effectively and efficiently uncover buffer overflow errors. This is an important requirement when adopting dynamic symbolic execution to test real world software systems [29], [35], [65]. In the next section, we elaborate our approach and emphasize its significance for improving active property checking for buffer overflow detection capabilities.

6.3 Approach

When attempting to mitigate buffer overflow vulnerabilities through automated techniques, the capability of static analysis approaches is limited when dealing with complex programming language constructs and concepts such as pointers, pointers arithmetic, and object-oriented polymorphic functions. In fact, the problem of statically verifying buffer overflows is, in general, undecidable. Furthermore, static heuristic solutions may not by applicable in practice. It has been recognized that despite the

considerable effort to statically and automatically tackle buffer overflow defects, in many cases concretely executing the program under test is the only way to address this problem [60], [115]. In our proposed approach, we use static analysis, particularly static runtime verification approaches such as DEPUTY [31] and CCURED [97], to pinpoint potential overflow vulnerabilities on buffer operations. The determination of whether a buffer overflow is truly vulnerable is done by dynamic analysis.

Dynamic analysis techniques for buffer overflow detection suffer from the following two main limitations: (1) the need for suitable test suites, i.e. test inputs capable of uncovering buffer overflows, and (2) the potential endlessness of the state space of the program under test. Dynamic symbolic execution is an effective test input generation technique. It offers a *directed automated* search mechanism to direct path exploration over the path space of the program in a desired manner. This feature can be used, in collaboration with static analysis, to actively search for specific paths leading to buffer overflows. This observation was crucial to the development of our approach.

To this end, we propose a goal-oriented testing approach, where a goal is a potential safety violation and the testing approach is to generate test inputs to uncover the violation. We used static runtime verification to diagnose potential safety violations and dynamic symbolic execution to perform test input generation. A major challenge in such an application of dynamic symbolic execution is the combinatorial explosion problem of the path space. To address this fundamental scalability issue, we take advantage of data dependence analysis to identify a root cause leading to the execution of the goal and use the chaining guided search algorithm GUIDER to effectively perform dynamic symbolic execution for exploring the test goal.

In the remainder of this section, we propose SEBO — a dynamic Symbolic Execution-based Buffer Overflow testing framework. The main focus of SEBO is to explore buffer overflow vulnerabilities in C programs by combining static runtime verification and dynamic symbolic execution techniques. The challenging problems motivating the development of SEBO can be summarized as follows:

- Static runtime verification can ensure memory safety by inserting runtime checks to verify potential overflow vulnerabilities on buffer operations. However, it lacks the ability to uncover circumstances under which buffer operations are vulnerable.

- Dynamic symbolic execution is an effective technique to automate test input generation. However, dynamic symbolic execution itself is not sufficient to trigger program bugs. Moreover, the path space explosion is problematic to apply this technique.

The execution mechanism of SEBO is therefore designed to first diagnose *potential* buffer overflow vulnerabilities using DEPUTY [31]—a novel type system for pointers—and then to explore *actual* vulnerabilities using CREST [19]—an extensible symbolic execution engine—to perform dynamic symbolic execution for test input generation. To enable this mechanism, SEBO consists of the following design components: buffer overflow checking, dynamic symbolic execution-based test input generation, and goal-oriented testing. In the following subsections, we describe these design components that comprise our proposed framework SEBO for buffer overflow vulnerability detection.

6.3.1 Buffer Overflow Checking

Given a C program under test, SEBO uses DEPUTY [31]—a novel type system for pointers, to enforce type and memory safety in the program. *Type safety* means that the runtime values of all memory locations correspond to their compile-time type, and *memory safety* means that all memory accesses are within the bounds of an allocated memory region. Enforcing these safety properties is an important step toward improving C programs as this eliminates many common security vulnerabilities, such as buffer overflows. In DEPUTY, all buffer operations are verified by a hybrid type-checking approach. In the first place, DEPUTY verifies buffer operations by *statically* reasoning about runtime values of expressions in the program. For buffer operations where static verification is not sufficient, DEPUTY enforces safety policies by inserting runtime checks. Any violations to runtime checks reveal errors in the program. Programmers can use runtime checks to check under which circumstances violations occur.

SEBO translates runtime checks into test goals, each test goal expresses the *opposed semantic* of a runtime check, e.g. by negating runtime check conditions. A test goal encodes conditions under which buffer operations are vulnerable. As a result, the goal of exploring buffer overflow errors can now be reduced to finding test inputs to explore test goals. To illustrate, consider the piece of code given in Figure 6.6(a), which assigns `'?'` symbol to `buf` array of size `SZ` (= 1000) at the index `index`. DEPUTY verifies this memory write operation may not be safe and inserts a runtime check right before this

```
if (index < 0 || index > SZ)
      return;
//BAD: overflow if index == SZ
buf [index] = '?';
```

(a) SOURCE CODE

```
if (index < 0 || index > SZ)
      return;
CLeq((buf+index)+1, buf+1000,
   "pointer access check",
   "(buf+index)+1 <= buf+1000",
   __LOCATION__);
buf [index] = '?';
```

(b) DEPUTY OUTPUT

```
if (index < 0 || index > SZ)
      return;

if (index >= 1000)
      __SEBO_ERROR();

buf [index] = '?';
```

(c) SEBO OUTPUT

Figure 6.6: Buffer overflow vulnerability example

write operation Figure 6.6(b). This check `CLeq` is a *pointer access check* with the condition `(buf + index) + 1 <= buf + 1000`, implying that the index to `buf` array must be less than its size. SEBO expresses this check in the meaning that if the index `index` to `buf` array is equal or greater than its size `(index >= 1000)`, then there is a memory access error `__SEBO_ERROR()`. Figure 6.6(c) shows SEBO's output.

Note, however, that runtime checks inserted by DEPUTY can be false positives, or be redundant with respect to checking memory safety failures. In DEPUTY, to eliminate redundant checks, type checking is followed by a flow-sensitive optimization phase. Specifically, it implements a global dataflow analysis for copy propagation and linear machine arithmetic to reason statically checks that are unnecessary. The quality of the optimizer, however, is limited due to complex constraints, data structures, and native calls of real world software. This is the intrinsic nature of static analysis techniques. SEBO does not attempt to prove redundant runtime checks; instead, it actively automatically finds test inputs to falsify runtime checks for buffer overflow detection.

6.3.2 Dynamic Symbolic Execution-Based Test Generation

SEBO uses CREST [19] — an extensible symbolic execution engine — to perform test input generation. After the buffer overflow checking phase, the output program (together with a set of test goals) is instrumented for concrete and symbolic execution. In CREST, one can configure search algorithms to carry out path exploration. The output is explored test goals and corresponding test inputs. We now illustrate our statement above that dynamic symbolic execution itself is not sufficient to trigger program bugs. In Figure 6.6(a), the given code adds three more paths into the path space, these being:

path 1: `index < 0`

path 2: `index >= 0 ∧ index > 1000`

path 3: `index >= 0 ∧ index <= 1000`

Obviously the execution of paths 1 and 2 is safe concerning the memory write operation through the `buf` array since the control flow of these two paths never can reach this memory access. The execution of path 3 necessitates the satisfiability of the constraint system: `index >= 0 ∧ index <= 1000`. For this, the underlying constraint solver yields one out of 1001 possible solutions to explore this path. That is, there is less than a 0.1% chance that the solution: `index = 1000` can be obtained to trigger the one-off-bound array access error in the code. Unsafe programming languages like C permit vulnerable memory operations to execute and dynamic symbolic execution alone is therefore not sufficient to expose memory defects.

In SEBO, as shown in Figure 6.6(c), the buffer overflow error can obviously be exposed by exploring `__SEBO_ERROR()` with condition `index >= 1000`. This condition, when associated with the path constraint of path 3, enforces the constraint solver to return the solution: `index = 1000` to uncover the overflow vulnerability. The use of DEPUTY is hence essential in detecting buffer overflow vulnerabilities in low level C programming.

6.3.3 Goal-Oriented Testing

The overall objective of SEBO is to strengthen the buffer overflow detection capability by combining static runtime verification and dynamic symbolic execution techniques. In SEBO, any search algorithm can be employed to perform path exploration for exploring buffer overflow errors. Recognizing the combinatorial explosion problem of path space experienced by dynamic symbolic execution, the SEBO design reduces the testing task to exploring given test goals in the code rather than exhaustively exploring all feasible program paths. This may lead to a significant reduction in the number of program paths that need to be explored. Our search algorithm GUIDER, proposed in this chapter 4, attempts to achieve this goal.

Figure 6.7 presents the logical components and the basic workflow of the SEBO framework. It also illustrates how SEBO extends the functionality of the DEPUTY framework to identify potential memory violations and the functionality of the CREST platform to automate test input generation for buffer overflow vulnerability testing. Basically, the architecture of SEBO is similar to that of the SCT structural coverage testing framework presented in chapter 5 (Figure 5.3). The two frameworks differ as to their purpose. SEBO aims to test security vulnerabilities while SCT seeks to improve

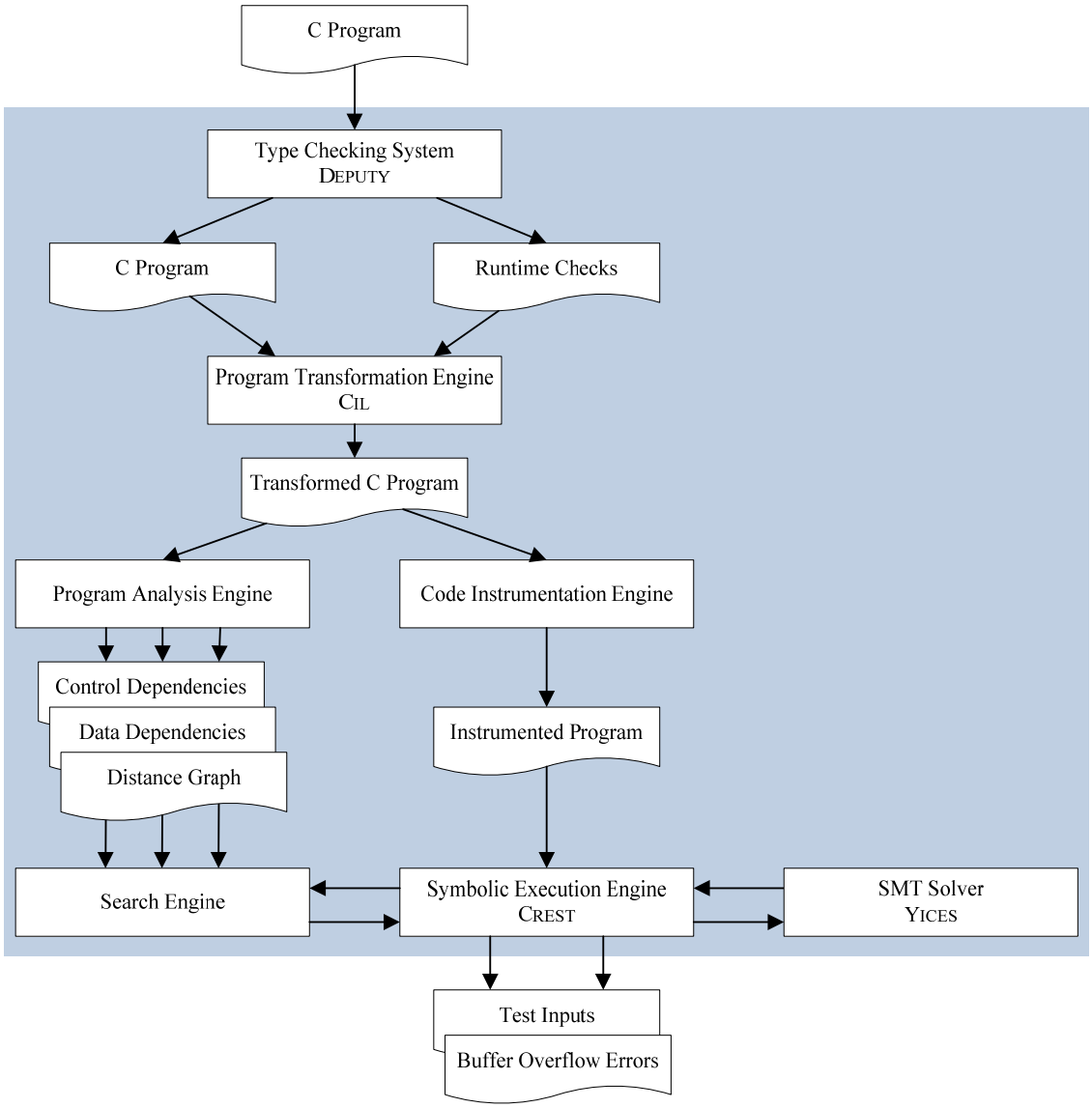

Figure 6.7: The proposed buffer overflow vulnerability testing framework

structural coverage results. The logical component *Search Engine* in SEBO performs path exploration for exposing overflow defects, not coverage elements.

As shown in Figure 6.7, the input to the SEBO framework is a C program under test P; and the framework first uses DEPUTY to enforce memory safety in the program. The output after this phase is another P' but type- and memory-safe C program, with a set of runtime checks inserted into P'. These runtime checks then are interpreted in forms of test goals, where the testing task is to active search algorithms to perform dynamic symbolic execution to expose test goals. This procedure is similar to that of the SCT structural coverage testing framework, where the *Search Engine* component interacts with *Symbolic Execution Engine* (CREST) and *SMT Solver* (YICES) components to

automatically perform test input generation. In order to effectively and efficiently carry out dynamic symbolic execution, SEBO employs GUIDER to guide path exploration. The output is a set of detected buffer overflow vulnerabilities detected, together with test inputs to witness the presence of these vulnerabilities.

In the implementation, SEBO extends DEPUTY [31] with approximately 500 lines of OCaml code to transform runtime checks generated by DEPUTY into test goals. We have also modified the logical component *Search Engine* to perform GUIDER for the purpose of exploring buffer overflows. In the next section, we present experiments conducted to evaluate the effectiveness of SEBO in testing buffer overflow defects for C programs.

6.4 Evaluation

The overall objective of this research project is to explore automated techniques in order to improve the efficiency of software testing. In this chapter, we have presented SEBO, our proposed approach for buffer overflow testing for C programs. SEBO has two objectives: (1) to mitigate the low precision of static analysis, particularly of static runtime verification for memory safety checking; (2) to optimize path exploration in dynamic symbolic execution for effectively and efficiently discovering buffer overflows. We now describe the evaluation conducted to assess how effective our proposed approach is in supporting buffer overflow testing. Specifically, we assess the effectiveness of SEBO by considering the following four research questions:

- In buffer overflow detection, how does SEBO perform in comparison with systematic depth-first search?
- In buffer overflow detection, how does SEBO perform in comparison with coverage-optimized search?
- How does the time setting influence the capability of search algorithms in buffer overflow detection?
- How does the number of paths explored influence the capability of search algorithms in buffer overflow detection?

In the following subsections, we present (1) the set of selected test subjects; (2) our evaluation methodology; (3) the experimental results; and (4) our discussion of the results. The purpose of having the above research questions in the evaluation is explained in section 6.4.2.

Table 6.1: An overview of the test subjects selected in the evaluation of the SEBO framework for buffer overflow testing

Program	Subject	Vulnerability	# Loc	# Statements	# Branches
apache	1	CVE-2004-0940	167	292	94
	2	CVE-2006-3747	184	270	48
edbrowse	3	CVE-2006-6909	161	179	38
gxine	4	CVE-2007-0406	58	47	6
libgd	5	CVE-2007-0455	318	386	96
MADWiFi	6	CVE-2006-6332	134	108	20
NetBSD-libc	7	CVE-2006-6652	166	200	42
OpenSER	8	CVE-2006-6749	160	266	62
	9	CVE-2006-6876	111	119	20
samba	10	CVE-2007-0453	102	68	8
SpamAssassin	11	BID-6679	57	91	16
bind	12	CA-1999-14	166	119	18
	13	CVE-2001-0011	362	299	66
wu-ftpd	14	CVE-1999-0368	278	383	66
	15	CVE-1999-0878	89	86	10
	16	CVE-2003-0466	257	351	62
sendmail	17	CVE-1999-0047	481	544	148
	18	CVE-1999-0206	108	97	22
	19	CVE-2001-0653	269	200	52
	20	CVE-2002-0906	66	71	10
	21	CVE-2002-1337	611	749	246
	22	CVE-2003-0161	115	104	30
	23	CVE-2003-0681	92	104	22
Total	—	—	**4512**	**5133**	**1202**

6.4.1 Test Subjects

We used the buffer overflow benchmark developed by Ku *et al.* [80] to conduct the evaluation. The benchmark consists of 23 buffer overflow vulnerabilities extracted from historic vulnerabilities in 12 widely used applications written in C programming language. Most of the vulnerabilities come from the Common Vulnerabilities and Exposures (CVE) database [37]. They cover a wide variety of overflow errors with sufficient complexity to make analysis challenging. Each vulnerability comes with a set of different types of programs, representing different levels of difficulties of

vulnerability exploration. We selected the most difficult programs of sizes from 50 to 650 lines of C code to perform the experiments. Table 6.1 gives an overview of these selected test subjects. Note that a program may have several test subjects, and each test subject represents a particular buffer overflow vulnerability. For example, the FTP server daemon *wu-ftpd* has three test subjects, these being 14, 15 and 16; and these test subjects corresponds to vulnerabilities *CVE-1999-0368*, *CVE-1999-0878* and *CVE-2003-0466*, respectively.

6.4.2 Methodology

Using the same set of test subjects, we compared the buffer overflow detection of our proposed testing framework SEBO with that of two widely adopted search algorithms: *depth-first search* (or DFS) and *control-flow graph directed search* (or CFGDIRECTED). DFS [62] offers a systematic search mechanism to exhaustively explore all feasible program paths and much research has confirmed its ineffectiveness in practical situations due to the combinatorial explosion of the path space. We therefore evaluated to what extent SEBO can mitigate this path explosion problem as suffered by DFS to explore overflow errors. In contrast to DFS, CFGDIRECTED [19] is an example of a coverage-optimized approach. This search algorithm makes use of the static control flow graph of the program to guide the search down short static paths to unexplored code. Burnim and Sen [19] showed that CFGDIRECTED was the best search algorithm to maximize structural coverage. We therefore evaluated SEBO against CFGDIRECTED since recent approaches tend to strengthen error detection capabilities by improving structural coverage, for instance KLEE [26] and SAGE [64].

It is important to notice that in the context of the SEBO framework, a potential buffer overflow error is encoded as a statement in the program. As a result, the task of testing for buffer overflow detects is actually reduced to exploring specific code elements of the program. Any path exploration algorithm can therefore be employed to perform dynamic symbolic execution for discovering buffer overflow vulnerabilities. We limited our evaluation to DFS and CFGDIRECTED, but we could have also compared SEBO with testing tools such as PEX [128] and FITNEX [141], which were used in the previous chapter for the evaluation of structural coverage testing. For this experimental benchmark, most test subjects involve intensive pointer usages that are not easy to convert into C# code. We also did not consider *random input search* (or RANDOM) in

this evaluation, due to its poor performance in test input generation as presented in the previous chapter.

For the time setting, we used 4 time limits (1 minute, 5 minutes, 10 minutes, and 30 minutes) and conducted the experiments using the following strategy. With a given search algorithm, we started with the first time limit (1 minute) and ran it on all test subjects, 1 minute for each subject. For each test subject, we captured: (1) if the buffer overflow vulnerability was uncovered; (2) the number of test goals explored; (3) the time amount spent; and (4) the number of paths explored. For all the test subjects that the search algorithm could not uncover vulnerabilities, we conducted the experiments again but on the next time limit (5 minutes). This procedure was repeated for the 10 and 30 minute time limits. By doing this, we could study the impact of time on the capability of search algorithms in exploring buffer overflow errors. Note that we stopped a search algorithm when it could explore all test goals of the current test subject and recorded the time used. If at least one test goal was explored, we considered the vulnerability to have been uncovered and did not move to larger time limits. This was done with the understanding that (1) if an error was detected, fixing it may prevent other errors to happen; and (2) due to the limitation of static runtime verification, not all test goals in SEBO represent real buffer overflow errors.

All experiments were run on 3GHz CoreTM2 Duo CPU with 4GB of RAM, running Ubuntu GNU/Linux with a 32-bit kernel version 3.2.0–38. In the SEBO approach, a test subject is first analysed to enforce memory safety by means of runtime checks. These runtime checks then were interpreted in the form of test goals, where the testing task was to activate search algorithms to perform dynamic symbolic execution to expose test goals.

6.4.3 Experimental Results

Table 6.2 summarizes the experimental results we obtained after the first testing time limit (1 minute). DFS was able to successfully uncover buffer overflow vulnerabilities for 13 of the 23 test subjects. Remarkably, for 10 of these uncovered vulnerabilities, DFS accomplished the path exploration process within 1 or 2 seconds. A manual investigation of the 10 subjects showed that all the test subjects had only one test goal and the structure of the program leading to the execution of the test goal was amenable to depth-first orders. This enabled DFS to uncover the test goal within a small number of

Table 6.2: Testing results of DFS, CFGDIRECTED, and SEBO when performed on 23 subjects in 1 minute

Program	Subject	DFS	CFGDIRECTED	SEBO
apache	1	N / (0/3) / 60 / 2741	Y / (3/3) / 5 / 1587	Y / (3/3) / 1 / 51
	2	N / (0/1) / 60 / 997	N / (0/1) / 60 / 22041	Y / (1/1) / 0 / 29
edbrowse	3	N / (0/1) / 60 / 668	N / (0/1) / 60 / 5874	Y / (1/1) / 1 / 36
gxine	4	Y / (1/1) / 0 / 17	N / (0/1) / 60 / 21747	Y / (1/1) / 0 / 17
libgd	5	N / (0/1) / 60 / 15843	Y / (1/1) / 5 / 1661	Y / (1/1) / 6 / 1050
MADWiFi	6	Y / (1/2) / 0 / 1	Y / (1/2) / 60 / 18057	Y / (1/2) / 0 / 1
NetBSD-libc	7	N / (0/1) / 60 / 1477	N / (0/1) / 60 / 19186	Y / (1/1) / 1 / 37
OpenSER	8	N / (0/1) / 60 / 13240	N / (0/1) / 60 / 19810	Y / (1/1) / 0 / 53
	9	N / (0/1) / 60 / 17792	N / (0/1) / 60 / 21221	Y / (1/1) / 0 / 23
samba	10	Y / (1/1) / 0 / 17	N / (0/1) / 60 / 21515	Y / (1/1) / 0 / 18
SpamAssassin	11	Y / (1/1) / 0 / 199	Y / (1/1) / 0 / 22	Y / (1/1) / 0 / 15
bind	12	Y / (1/1) / 0 / 2	Y / (1/1) / 0 / 6	Y / (1/1) / 0 / 4
	13	Y / (1/1) / 0 / 30	N / (0/1) / 60 / 20820	Y / (1/1) / 0 / 33
wu-ftpd	14	Y / (1/1) / 0 / 40	N / (0/1) / 60 / 19780	Y / (1/1) / 1 / 50
	15	Y / (1/1) / 1 / 28	N / (0/1) / 60 / 21688	Y / (1/1) / 0 / 28
	16	Y / (1/1) / 1 / 56	N / (0/1) / 60 / 20683	Y / (1/1) / 3 / 661
sendmail	17	Y / (2/7) / 60 / 8693	Y / (4/7) / 60 / 12855	Y / (4/7) / 60 / 5254
	18	Y / (2/3) / 60 / 3270	N / (0/3) / 60 / 21700	Y / (2/3) / 60 / 7003
	19	N / (0/1) / 60 / 11860	N / (0/1) / 60 / 18239	N / (0/1) / 60 / 3360
	20	Y / (1/1) / 0 / 4	Y / (1/1) / 0 / 4	Y / (1/1) / 0 / 5
	21	N / (0/28)/ 60 / 647	N / (0/28)/ 60 / 2841	Y / (1/28)/ 60 / 1483
	22	N / (0/3) / 60 / 1908	N / (0/3) / 60 / 11417	Y / (2/3) / 60 / 2620
	23	Y / (1/2) / 60 / 10685	N / (0/2) / 60 / 20342	Y / (2/2) / 0 / 42

Note: The testing result has the format: [Y | N] / (X/Y) / T / P, where Y means the vulnerability was uncovered and N otherwise, X is the number of test goals explored, Y is the total number of test goals of the test subject, T is the time amount spent (measured in seconds), and P is the number of paths explored.

path explorations. For the other 3 cases, DFS timed out while attempting to uncover all the test goals of each test subject.

CFGDIRECTED uncovered 7 of 23 buffer overflow vulnerabilities, demonstrating a less effective performance than DFS for the first testing time limit. Amongst these 7 cases, CFGDIRECTED was able to terminate the path exploration process within 5 seconds for 5 cases; for the other 2 cases, it timed out. Noticeably, for several test subjects where DFS could discover overflow errors within a matter of a few seconds, CFGDIRECTED could

not even though it explored a considerable number of paths. CFGDIRECTED algorithm failed to uncover 16 cases within 1 minute of testing.

SEBO demonstrated a significantly improved performance over both DFS and CFGDIRECTED. It discovered in total 22 buffer overflow vulnerabilities. The only case that SEBO failed to uncover after one minute of testing was *sendmail/CVE-2001-0653*. Amongst the 22 cases uncovered, SEBO timed out in 4 cases while attempting to explore all test goals. For the other 18 cases, SEBO optimized path exploration to uncover vulnerabilities within relatively small numbers of paths explored. In the case of w*u-ftpd/CVE-2003-0466*, DFS took 1 second and explored 56 paths to discover the vulnerability, while SEBO spent 3 seconds and explored 661 paths to achieve the same result.

In summary, after the first testing time limit, DFS failed in 10 cases, CFGDIRECTED 16 cases, and SEBO just 1 case to uncover buffer overflow vulnerabilities. We continued the experiments on these failing cases with extended testing time limits. The purpose this was to determine if the capability of search algorithms in detecting buffer overflow vulnerabilities is improved when the testing time is expanded through the ability to explore many more program paths. We started with 5 minutes, moved to 10 minutes, and finally 30 minutes. In the consideration of the relatively small size of the selected test subjects, we decided to stop after 30 minutes. The experimental results are given in Tables 6.3–6.5. Note that subjects in "Subject" columns refer to the corresponding vulnerabilities in Table 6.1 and the figures are the numbers of paths explored by search algorithms within given testing time limits.

After 5 minutes of testing, DFS (Table 6.3) timed out in all 10 cases. On *sendmail/ CVE-2003-0161* subject, it was able to uncover the buffer overflow vulnerability, however. When the 9 remaining failing cases were tested on the extended time limit (10 minutes), none could be explored by this search algorithm. When the time limit was set to 30 minutes, DFS discovered the buffer overflow error of *libgd/CVE-2007-0455* only after 606 seconds ,with 146,631 paths explored.

For CFGDIRECTED (Table 6.4), of the 16 failing cases, only one, *sendmail / CVE-2002-1337*, was uncovered after 5 minutes. For the remaining 15 cases, on both the 10 minute and 30 minute time limits, CFGDIRECTED was not able to uncover any cases. Remarkably, with increases in the testing time we witnessed a massive increase in the

Table 6.3: Numbers of explored paths by DFS on 10 subjects after 5 minutes, 10 minutes and 30 minutes of testing

Program	Subject	DFS		
		5 minutes	10 minutes	30 minutes
apache	1	8229	16348	47779
	2	1805	2568	5705
edbrowse	3	3159	6114	18299
libgd	5	74053	144913	**146631**
NetBSD-libc	7	4985	9354	25168
OpenSER	8	63037	123228	357298
	9	82795	162538	466363
sendmail	19	56872	108805	313130
	21	2497	4522	25496
	22	**2999**	—	—

Table 6.4: Numbers of explored paths by CFGDIRECTED on 16 subjects after 5 minutes, 10 minutes and 30 minutes of testing

Program	Subject	CFGDIRECTED		
		5 minutes	10 minutes	30 minutes
apache	2	105970	209983	621714
edbrowse	3	9793	19561	108802
gxine	4	105717	208689	621818
NetBSD-libc	7	91535	179648	549749
OpenSER	8	93380	185361	587953
	9	98522	200057	598077
samba	10	105025	211813	623439
bind	13	96970	198744	574827
wu-ftpd	14	92692	186849	563949
	15	101617	205735	614851
	16	94752	196361	561692
sendmail	18	99612	204932	607686
	19	85548	172897	545163
	21	**19227**	—	—
	22	56374	114811	350075
	23	97270	194751	597552

Table 6.5: Numbers of explored paths by SEBO on 1 subject after 5 minutes, 10 minutes and 30 minutes of testing

Program	Subject	SEBO		
		5 minutes	10 minutes	30 minutes
sendmail	19	8424	10244	10348

number of program paths explored by both DFS and CFGDIRECTED. However, looking for a path to trigger the execution of buffer overflow errors is problematic for these search algorithms even though we were testing programs with a few hundreds of lines of code.

SEBO (Table 6.5) failed to uncover the buffer overflow vulnerability of its only failed case in all time limits. Note however that on this test subject both DFS and CFGDIRECTED also failed. This is because on this test subject, *sendmail/CVE-2001-0653*, DEPUTY is able to trigger the buffer overflow error but it is not able to trigger the *integer underflow* error, which is the root cause leading to the buffer overflow. Dynamic symbolic execution that only focuses on path exploration is not sufficient to uncover this buffer overflow vulnerability.

6.4.4 Discussion

We conclude the experiments with the following observations. First, our proposed buffer overflow testing framework SEBO, in combination with our proposed chaining guided search algorithm GUIDER, demonstrated a significant improvement over both DFS and CFGDIRECTED in both the capability to uncover buffer overflow errors and the capability to optimize the path exploration. For several test subjects, SEBO could explore the buffer overflow errors within a matter of a few seconds, while DFS and CFGDIRECTED both failed even after 30 minutes of testing. This illustrates the efficiency of using data dependence analysis in guiding dynamic analysis. Second, while the time setting does influence the capability of search algorithms in buffer overflow detection, its impact is relatively small. Finally, exploring more paths has very little impact on improving the buffer overflow detection capability.

We evaluated our testing framework using only one benchmark of 23 relatively small-sized programs. The evaluation setting used two baseline search algorithms and four testing time limits. It is possible that other programs and other evaluation settings would

exhibit different results, and in the future we intend to extend our proposed approach and conduct experiments on large programs to properly assess the validity of our proposal and observations. We believe that when testing sizeable and complex programs where the path space is too large to exhaustively explore, our proposed approach's ability to break down the path space and to precisely guide the path exploration by focusing on selected aspects of semantics is essential for optimizing the very expensive cost of performing dynamic symbolic execution to strengthen security vulnerability detection.

6.5 Summary

Structural coverage criteria offer useful software measures to assess the adequacy of software testing. Enforcing test adequacy is an attempt to improve the quality and reliability of software upon deployment and the confidence that users can have in it. In the context of the potential infiniteness of the state space, the use of structural coverage criteria is not adequate for detecting software defects. Furthermore, it is not possible to prove the correctness of software. In practice, proposed testing techniques tend to maximize the capability of detecting software defects by evaluating the software against various quality factors such as functionality, performance, reliability, and security.

Security has recently become a very serious issue in the practice of software development [63], [122]. Approximately 50% of security vulnerabilities are introduced at the source code level [102] and buffer overflows account for nearly half of all known security vulnerabilities in real world software [135]. Our proposed buffer overflow testing framework SEBO has been developed to improve the capability of automated software testing for security vulnerability detection. Perhaps, the most distinguishing feature of SEBO is the use of our proposed chaining guided search algorithm GUIDER, which is guided by data dependency analysis and is based on dynamic symbolic execution to significantly enhance the efficiency of path exploration for automatically exploring memory safety violations in C programs. GUIDER has been shown to be more effective and efficient than standard search algorithms such as DFS and coverage-optimized such as CFGDIRECTED in dealing with the path space explosion problem of dynamic symbolic execution.

In order to perform buffer overflow checking, dynamic symbolic execution-based techniques such as EXE [30] and SAGE [64] systematically inject assertions into the program during test input generation. This may lead to a performance burden on the entire testing system, which is already slow because of the expense of performing dynamic symbolic execution and the unpredictable effectiveness of constraint solvers. PEX [128] and KLEE [26] work on .NET Framework and LLVM [83], respectively, where memory safety is verified by the underlying compiler. In our SEBO testing framework, DEPUTY [31] was utilized to provide a goal-oriented testing approach, such that dynamic symbolic execution was used for uncovering buffer overflow vulnerabilities instead of systematically exploring all possible program paths.

Finally, the preliminary evaluation conducted against 23 buffer overflow vulnerabilities showed a significant improvement of our search algorithm over two popular state-of-the-art search algorithms in both the capability to uncover vulnerabilities and the capability to optimize path exploration. The evaluation also provides valuable observations in the context of developing techniques to perform dynamic symbolic execution for uncovering buffer overflow vulnerabilities.

Chapter 7

Conclusion

Software pervades every aspect of our life: businesses, financial services, medical services, communication systems, entertainment, and education are largely dependent on software. With this increasing dependency on software, we expect software to be reliable, robust, safe and secure. Unfortunately, the reliability of everyday software is questionable. Software failures are reported daily together with substantial financial consequences and reputational damage to businesses, as well as stress and inconvenience to customers. In the practice of software development, although much progress has been made in software verification and validation, software testing remains one of the primary ways widely adopted to improve the reliability of software. Often half of the total software development costs are devoted to testing. Even after such a huge investment, serious defects and security breaches are common in widely used and well-tested software. This is partly because testing software is mostly manually performed, and thus expensive and unreliable.

In this thesis, we explored and developed effective methods to improve the degree of the attainable software testing automation. In particular, this thesis looked closely at methods to effectively automate test input generation, a fundamental activity of software testing. These methods use ideas from symbolic analysis, constraint solving, dynamic program analysis, control and data dependence analysis, and static runtime verification, and apply them to build effective testing frameworks to strengthen testing objectives such as structural coverage testing and security testing. This final chapter summarizes the development of this thesis, discusses the issues encountered, and suggests possible future directions for research in light of this study's findings.

7.1 Summary

Chapter 1 of the thesis introduced the motivations for undertaking this research, defined the study scope, and finally outlined the thesis structure. This research was primarily focused on improving the efficiency of software testing by exploring and developing techniques to effectively and efficiently automate the test input generation process. The testing objectives explored in this study were structural coverage testing and security vulnerability testing. We also highlighted the contribution of our study to the existing body of scientific knowledge as well as the research publications that have been generated by this research.

In chapter 2, after illustrating the function of testing in software quality assurance, testing activities, and testing throughout the life cycle of software development, we emphasized the need for automated software testing to improving the quality and reliability of software. The motivating force for the development of automated testing is the intensive intervention of humans in every part of the software testing process which makes testing laborious, expensive, and unreliable.

The chapter continued with an extensive literature review of automated test input generation techniques. Specifically, we reviewed random testing, symbolic execution, dynamic symbolic execution, search-based testing, and the chaining approach—the five main automated test input generation techniques proposed over the last three decades. We evaluated these techniques by comparing and contrasting their strengths and shortcomings in handling large size and high complexity of real world software systems.

Among the above techniques, dynamic symbolic execution has been shown to be an effective technique to automate test input generation. It intertwines the strengths of random testing and symbolic execution to achieve the scalability and high precision of dynamic analysis and the power of the underlying constraint solver. We thus chose dynamic symbolic execution as an enabling technology for the development of this thesis. The fundamental scalability challenge facing dynamic symbolic execution is the combinatorial explosion of the path space. The development of this thesis involved exploring and developing methods to address the path explosion problem in dynamic symbolic execution to improve testing objectives such as structural coverage testing and

security testing. Chapter 2 concluded by presenting the three main objectives of this research.

We presented the theoretical background of dynamic symbolic execution in chapter 3. This technique couples concrete and symbolic execution to explore the path space of a program. Concrete execution enables symbolic execution to alleviate the effects of the incompleteness of the underlying reasoning engines. For instance, concrete execution helps simplify the constraints that theorem provers cannot handle, resolves aliases for pointers, and handles the unavailability of source code. On the other hand, symbolic execution helps establish an automated directed exploration mechanism to explore the program path space. It generates test inputs that lead the program to different concrete executions. The biggest scalability challenge facing dynamic symbolic execution was also highlighted in this chapter. An attempt to mitigate this challenge is to develop effective techniques to prioritize path exploration in order to achieve the best results of specific testing objectives.

A goal-oriented dynamic test generation approach was proposed in chapter 4, where a goal is a code element, e.g. statement or branch, in the program and the testing approach attempts to find test inputs to uncover the goal. This is referred to as the reachability problem in computer science. We proposed using the chaining approach to guide path exploration in dynamic symbolic execution. Given a test goal to explore, the chaining approach first performs data dependence analysis to identify statements that affect the execution of the test goal and then uses these statements to create sequences of events that are to be executed prior to the execution of the test goal. The advantage of doing this is three-fold: (1) it precisely focuses on getting the test goal to be executed; (2) it forms a search mechanism to effectively perform the path exploration process; and (3) it slices away code segments that are irrelevant to the execution of the test goal. These three strengths together form a search mechanism to guide the path exploration process into potentially unexplored but promising areas of the program path space to unclose high-complexity code.

Based on the chaining approach, we proposed the search algorithm GUIDER. GUIDER is driven by the chaining mechanism and is based on dynamic symbolic execution to perform path exploration for exploring the test goal. GUIDER distinguishes itself from existing search algorithms in three major aspects: (1) it mitigates the path explosion

problem by centralizing on data dependences which truly affect the executability of the test goal; (2) it is able to refine path exploration when the local search space is saturated; and (3) it determines control dependences on the fly and exploits the static program structure to optimize path exploration.

Based on the goal-oriented dynamic test generation approach proposed in chapter 4, we proposed the structural coverage testing framework SCT in chapter 5. In SCT, a goal is a coverage element of the program under test and the testing approach is to find test inputs to uncover the test goal. SCT attempts to obtain the branch coverage criterion for the program under test. SCT was implemented based on the CREST platform, an extensible symbolic execution engine. We then evaluated SCT against 15 test subjects, both simulated and real world ones, and compared it with 5 different search algorithms widely adopted in research community. The experimental results demonstrated the capability of SCT to both effectively improve branch coverage results and efficiently optimize path exploration to uncover code elements that the other search strategies could not.

Structural coverage criteria have long been advocated in the software industry to assess the adequacy of software testing [20], [56], [117], [132]. Because exhaustively testing software is infeasible, an adequacy criterion can be considered as a stopping rule for determining whether sufficient testing has been carried out for it to be terminated. Test adequacy criteria can also provide measurements of test quality where a degree of adequacy associated with a test set acts as a level of confidence about the correctness of the software under test. A high coverage degree obtained implies that the program is more thoroughly tested and has a lower chance of containing software defects than a program with low structural coverage. Therefore, ensuring high structural coverage results is an important goal of software testing and our proposed framework SCT represents an attempt to achieve this goal.

In chapter 6, the security vulnerability testing framework SEBO was proposed to test buffer overflow vulnerabilities. In SEBO, a goal is a potential safety violation and the testing approach attempts to find test inputs to uncover violations. We used static runtime verification (or DEPUTY [31]) to diagnose potential safety violations and dynamic symbolic execution to perform test input generation. We evaluated SEBO against 23 buffer overflow vulnerabilities and compared it to two widely adopted search

strategies, namely *depth-first search* (DFS) and *control flow graph-directed search* (CFGDIRECTED). We observed a significant improvement of our search algorithm over both DFS and CFGDIRECTED in both the capability to uncover buffer overflow errors and the capability to optimize path exploration. For several test subjects, SEBO could explore the buffer overflow errors within a matter of a few seconds, while DFS and CFGDIRECTED both failed even after 30 minutes of testing.

The development of our security vulnerability testing framework SEBO represents an attempt to improve several aspects in software testing. Firstly, software testing is primarily focused on finding defects and about 55% of all defects are introduced during programming [67], the phase in which actual coding takes place. Secondly, structural coverage criteria provide useful measures to guide software testing; however, they do not provide a direct mechanism to discover software defects. Thirdly, testing security vulnerabilities such as buffer overflow often goes beyond the capability of traditional functional testing [137]. And last but not least, while researchers have explored and developed techniques to perform dynamic symbolic execution for error detection, improving structural coverage results does not directly imply an improvement in the capability to detect errors, especially subtle deep errors such as buffer overflow vulnerability defects. In our experiments, we showed that extending testing time limits and even exploring more program paths had little impact on uncovering overflow defects. The experiments provided valuable findings in which SEBO demonstrated its capability in improving the efficiency of testing security vulnerability defects.

7.2 Limitations of Our Work

This thesis has explored and developed techniques to improve the solving of the reachability problem. The problem statement was simple: given a test goal (e.g. statement or branch) in the program, find test inputs to uncover the test goal. The starting point was a test input generation technique — dynamic symbolic execution — and the primary objective was to explore methods to handle the path explosion problem for effectively and efficiently exploring specific test goals.

From the viewpoint of program comprehension, the execution of a test goal may be affected by only specific code segments in the program and so we can slice away code segments that are irrelevant to the execution of the test goal. This is referred to as a

program slicing technique [131] and can be used to perform path exploration for exploring test goals [113], [123]. Program slicing takes into account both control and data dependencies, and thus the resulting slice may be too large. It is important to note that even a single small piece of code can yield a number of paths that is too huge to be exhaustively explored. The approach proposed in this thesis was to consider only data dependencies and to adapt control dependencies on the fly. This was implemented in the GUIDER search algorithm to perform path exploration in dynamic symbolic execution. However, this approach does exhibit limitations inherent in both static and dynamic program analysis.

In the path exploration strategy implemented in GUIDER, when the search algorithm is given a test goal to explore, it first performs data dependence analysis to identify statements that affect the execution of the test goal, and then guides path exploration to propagate the effect of these statements up to the goal structure in order to trigger the execution of the test goal. The efficiency of this path exploration strategy heavily depends on the results of data dependence analysis, where pointers and calling contexts are the main sources leading to low precision analysis. In the implementation, we used a worklist-based dataflow algorithm, which is similar to the worklist algorithm presented in the work of Atkinson and Griswold [2], to compute data dependencies. This algorithm employs the Generalized One Level Flow (GOLF) approach [45] to perform pointer analysis. GOLF is, in general, a flow-insensitive context-insensitive algorithm, which means the control-flow structure of the program is not considered and calling contexts are not distinguished. As a result, GOLF can maintain scalability but results in low precision pointer analysis. The dataflow algorithm to compute data dependencies, however, is flow-sensitive but context-insensitive. In other words, the data dependence analysis implemented in GUIDER is not precise. Currently, GUIDER can perform well at unit testing level, or testing individual procedures without invocations. Extending it to test whole programs would require significant improvements in pointer analysis as well as dataflow analysis to accurately compute data dependencies. This is a significant limitation in the development of the GUIDER algorithm which limits the applicability of the testing frameworks SCT and SEBO.

GUIDER uses dynamic symbolic execution to perform test input generation and in both the testing frameworks SCT and SEBO we extended the CREST platform [19] to perform dynamic symbolic execution. Dynamic symbolic execution has its inherent limitations

due to the fact that symbolic execution, constraint generation, and constraint solving engines are necessarily imprecise. Specifically, dynamic symbolic execution cannot precisely handle floating point data type variables and non-linear arithmetic operations such as multiplication, division, modular, and bitwise operations. The unavailability of source code due to the presence of library calls or third-party components deteriorates symbolic execution into concrete execution. Pointers, function pointers, and symbolic offsets do complicate dynamic symbolic execution. Furthermore, CREST is bounded due to its own implementation limitations. For instance, CREST does not handle pointer inputs whereas CUTE [124] does; and CREST by using YICES SMT [47] solver does not deal with non-linear arithmetic theory while PEX [128] by using Z3 [89] does support specific non-linear arithmetic constraints. In order to tackle these limitations, dynamic symbolic execution downgrades to a partial form of symbolic execution and random testing. In the worst case, dynamic symbolic execution becomes a complete form of random testing, where dynamic symbolic execution is no longer able to utilize the strengths of symbolic analysis and constraint reasoning engines. Both SCT and SEBO, which extend CREST to implement dynamic symbolic execution, therefore suffer significantly from these limitations.

7.3 Future Work

The primary objective of this research was to explore and develop effective techniques to improve the efficiency of software testing. The starting point was the automated test input generation technique dynamic symbolic execution. We exploited program dependencies (e.g. control and data dependencies) to strengthen path exploration over the potentially infinite program path space for structural coverage testing as well as security vulnerability testing. However, currently our proposed approach is limited due to a number of factors, some of which result from the adoption of dynamic symbolic execution as discussed above. This section suggests future research work that could improve our current approach.

First of all, improving the precision of pointer analysis is an important step toward improving the effectiveness of our approach; high precision pointer analysis improves the accuracy of data dependence computation. A flow-sensitive context-sensitive pointer analysis such as that proposed by Landi and Ryder [85], Wilson and Lam [136], or Chatterjee, Ryder, and Landi [34] could be adopted to replace the current flow-

insensitive context-insensitive GOLF approach [45]. In addition, researchers could explore how systematic dynamic test generation can be intertwined with dynamic points-to analysis [88] to mitigate over-approximations caused by static pointer analysis. Precisely computing data dependences would greatly support our proposed search algorithm.

Dynamic symbolic execution is limited due to the effects of the incompleteness of the underlying reasoning engines. Alleviation of these effects could be achieved by employing more powerful SMT solvers to support more theories. For instance, replacing the YICES solver [47] with Z3 [89] would allow CREST to handle non-linear multiplication, division, and modular operations [144]. Other research might involve using search-based testing approaches to deal with floating point data type variables [84]. And for pointers and function pointers, we could even implement pointer solvers to analyze programs where input can be pointers [124], which go beyond the capability of current SMT solvers.

Additionally, while the development of path exploration strategies is crucial to improving the effectiveness of dynamic symbolic execution, dealing with sizable and complex real world software can require considerable computational power. Parallelism has been exploited to capacitate dynamic symbolic execution to testing industrial applications [66]. In the context of goal-oriented testing, parallelism can positively potentially enhance the efficiency of software testing by employing distinct parallel tasks to explore distinct test goals.

At this stage, the development of the testing frameworks SCT and SEBO is limited to structural coverage testing and security vulnerability testing. In security testing, we mainly concentrated on buffer overflow detection. Future work could explore further applications of our goal-oriented approach to test other types of errors and application domains [41], and other research directions such as regression testing [1], [107] and automated debugging [110].

7.4 Final Thoughts

I commenced my PhD studies in November 2010. In the first six months, I mainly focused on model checking techniques. I identified that model checking is mostly applicable to verify hardware and protocol designs, and the result of these initial

investigations was a survey paper accepted for the 6th International Conference on Evaluation of Novel Approaches to Software Engineering.

I then researched automated software testing and conducted an extensive literature review to assess the current state of research with a special focus on automated test input generation techniques. Amongst the techniques reviewed, dynamic symbolic execution, with its ability to intertwine the strengths of random testing, symbolic execution, and constraint solving engines, appeared to hold much promise for automating software testing. The fundamental scalability challenge limiting dynamic symbolic execution is the combinatorial explosion of the path space. To deal with this scalability issue, techniques such as compositional analysis [58] and abstraction [5] offer theoretical models to achieve completeness of path-based analysis. An alternative approach is to develop search heuristics to improve the efficiency of path exploration instead [35] and our approach proposed in this thesis fits into this research context.

I observed that the execution of a code element can be triggered by propagating appropriate data flows down to the target structure. I extended the chaining approach [52] and developed the search algorithm GUIDER to perform dynamic symbolic execution. The two testing frameworks SCT (for structural coverage testing) and SEBO (for buffer overflow vulnerability testing) were proposed based on GUIDER. During the conducting of the experiments, I realized that without an effective path exploration strategy, dynamic symbolic execution is not sufficient for testing real world software systems. For instance, in the experiments to evaluate SEBO against 23 buffer overflow vulnerabilities, dynamic symbolic execution with depth-first search and coverage-optimized search both failed to discover several vulnerabilities even after 30 minutes of testing on several test subjects.

The search algorithm GUIDER utilizes control and data dependencies to guide path exploration. Data dependence computation however is limited due to the flow-insensitive context-insensitive pointer analysis. In fact, context-sensitivity is the most challenging research problem for the program analysis community. I strongly believe that with sufficient research effort to improve pointer analysis, we can improve the effectiveness of the approaches proposed in this thesis, which are currently limited to unit testing.

References

[1] T. Apiwattanapong, "Identifying Testing Requirements for Modified Software," *Ph.D. dissertation, Georgia Institute of Technology*, Jul. 2007.

[2] D. C. Atkinson and W. G. Griswold, "Implementation Techniques for Efficient Data-Flow Analysis of Large Programs," in *ICSM*, 2001, pp.52.

[3] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random Testing: Theoretical Results and Practical Implications," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, Mar. 2012, pp. 258–277.

[4] P. Ammann and J. Offutt, "Introduction to Software Testing," *1st Edition, Cambridge University Press*, Jan. 2008.

[5] S. Anand, C. S. Păsăreanu, and W. Visser, "Symbolic execution with abstract subsumption checking," in *SPIN*, 2006, pp. 163–181.

[6] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF-SE: a symbolic execution extension to Java PathFinder," in *TACAS*, 2007, pp. 134-138.

[7] B. Beizer, "Software Testing Techniques," *2nd Edition, Itp – Media*, Jun. 1990.

[8] A. Bertolino, "ISSTA 2002 panel: is ISSTA research relevant to industrial users?" in *ISSTA*, 2002, pp. 201–202.

[9] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," in *Future of Software Engineering*, 2007, pp. 85–103.

[10] B. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List", *Computer Journal, IEEE Computer Society Press*, vol. 34, no. 1, Jan. 2001, pp. 135–137.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Communications of the ACM*, vol. 53, no. 2, 2010, pp.66-75.

[12] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Enhancing structural software coverage by incrementally computing branch executability," *Software Quality Journal*, vol. 19, pp. 725–751, Dec. 2011.

[13] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Symbolic execution with separation logic," in *APLAS*, 2005, pp. 52-68.

[14] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT—a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the international conference on Reliable software*, 1975, pp. 234-245.

[15] E. Bounimova, P. Godefroid, and D. A. Molnar, "Billions and billions of constraints: whitebox fuzz testing in production," in *ICSE*, 2013, pp. 122–131.

[16] D. Binkley, M. Harman, and K. Lakhotia, "FlagRemover: a testability transformation for transforming loop-assigned flags," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 12, Aug. 2011.

[17] D. L. Bird and C. U. Munoz, "Automatic generation of random self-checking test cases," *IBM Systems Journal*, vol. 22, no. 3, 1983, pp. 229-245.

[18] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software—Practice & Experience*, vol. 30, no. 7, Jun. 2000, pp. 775-802.

[19] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE*, 2008, pp. 443–446.

[20] British Standards Institute, "BS 7925-1 Vocabulary of Terms in Software Testing," 1998

[21] Calleam Consulting Ltd, "Denver Airport Baggage System Case Study," Retrieved from: http://calleam.com/WTPF/wp-content/uploads/articles/DIABaggage.pdf.

[22] ComputingCases.org, "A History of the Introduction and Shut Down of Therac-25," Retrieved from: http://computingcases.org/case_materials/therac/case_history/Case%20History.html.

[23] CREST, "Automatic Test Generation Tool for C," Retrieved from: http://code.google.com/p/crest/.

[24] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. 2, pp. 215–222, May 1976.

[25] P. D. Coward, "Symbolic Execution Systems - A Review," *Software Engineering Journal*, vol. 3, no. 6, Nov. 1988, pp. 229-239.

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

[27] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem," *IEE Proceedings - Software*, vol. 150, no. 3, Jun. 2003, pp. 161-175.

[28] C. Cadar and D. Engler, "Execution generated test cases: how to make systems code crash itself," in *SPIN*, 2005, pp. 2-23.

[29] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *ICSE*, 2011, pp. 1066–1071.

[30] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *ACM Transactions on Information and System Security*, vol. 12, no. 10, Dec. 2008.

[31] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *ESOP*, 2007, pp. 520–535.

[32] B. Chess and G. McGraw, "Static Analysis for Security," *IEEE Security and Privacy*, vol. 2, no. 6, Nov. 2004, pp. 76-79.

[33] CNN, "NASA's metric confusion caused Mars orbiter loss," Sep. 1999, Retrieved from: http://edition.cnn.com/TECH/space/9909/30/mars.metric/.

[34] R. Chatterjee, B. G. Ryder, and W. A. Landi, "Relevant context inference," in *POPL*, 1999, pp. 133-146.

[35] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, pp. 82–90, Feb. 2013.

[36] C. Csallner and Y. Smaragdakis, "Check 'n' crash: combining static checking and testing," in *ICSE*, 2005, pp. 422-431.

[37] Common Vulnerabilities and Exposures, http://cve.mitre.org/.

[38] Common Vulnerabilities and Exposures, "CVE-2006-5994," Retrieved from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0466.

[39] Common Vulnerabilities and Exposures, "CVE-2006-5994," Retrieved from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5994.

[40] Common Weakness Enumeration, http://cwe.mitre.org/.

[41] Common Weakness Enumeration, "The 2011 CWE/SANS Top 25 Most Dangerous Software Errors," Retrieved from: http://cwe.mitre.org/top25/.

[42] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," in *Proceedings of DARPA Information Survivability Conference and Exposition*, 2000, pp. 119-129.

[43] E. W. Dijkstra, "Chapter I: Notes on structured programming", in *Structured Programming, Academic Press Ltd. London*, 1972, pp. 1–82.

[44] D. Dean, E. W. Felten, and D. S. Wallach, "Java Security: From HotJava to Netscape and Beyond," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996, pp. 190.

[45] M. Das, B. Liblit, M. Fähndrich, and J. Rehof, "Estimating the Impact of Scalable Pointer Analysis on Optimization," in *SAS*, 2001, pp. 260-278.

[46] X. Deng, J. Lee, and Robby, "Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems," in *ASE*, 2006, pp. 157-166.

[47] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *CAV*, 2006, pp. 81-94.

[48] M. Dowd, J. McDonald, and J. Schuh, "The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities," *1st Edition, Addison-Wesley Professional*, Nov. 2006.

[49] Ú. Erlingsson, "Low-level software security: attacks and defenses," *Foundations of security analysis and design IV*, 2007, pp. 92-134.

[50] D. Engler and D. Dunbar, "Under-constrained execution: making automatic code destruction easy and scalable," in *ISSTA*, 2007, pp. 1-4.

[51] Fujitsu Limited, "Fujitsu Develops Technology to Enhance Comprehensive Testing of Java Programs," Jan. 2010, Retrieved from: http://www.fujitsu.com/global/news/pr/archives/month/2010/20100112-02.html.

[52] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering and Methodology*, vol. 5, Jan. 1996, pp. 63–86.

[53] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *PLDI*, 2002, pp. 234-245.

[54] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *WSS*, 2000, pp. 6–6.

[55] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, Jul. 1987, pp. 319-349.

[56] P. G. Frankl and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, Oct. 1988, pp. 1483-1498.

[57] S. Garfinkel, "History's Worst Software Bugs," in *Wired Magazine*, Oct. 2005, Retrieved from: http://www.wired.com/news/technology/bugs/0,2924,69355,00.html

[58] P. Godefroid, "Compositional dynamic test generation," in *POPL*, 2007, pp. 47–54.

[59] H. Goldstein, "Who Killed the Virtual Case File?" in *IEEE Spectrum*, Sep. 2005, Retrieved from: http://spectrum.ieee.org/computing/software/who-killed-the-virtual-case-file.

[60] C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting buffer overflow via automatic test input data generation," *Computers and Operations Research*, vol. 35, no. 10, Oct. 2008, pp. 3125-3143.

[61] A. Groce, G. Holzmann, and R. Joshi, "Randomized Differential Testing as a Prelude to Formal Verification," in *ICSE*, 2007, pp. 621-631.

[62] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[63] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson, "2006 CSI/FBI Computer Crime and Security Survey," *Computer Security Institute*, 2006, Retrieved from: http://gocsi.com/sites/default/files/uploads/FBI2006.pdf.

[64] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[65] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Active property checking," in *EMSOFT*, 2008, pp.207–216.

[66] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, Mar. 2012, pp. 40–44.

[67] D. Graham, E. V. Veenendaal, I. Evans, and R. Black, "Foundations of Software Testing: ISTQB Certification," *Revised Edition, Cengage Learning EMEA*, Jan. 2008.

[68] K. V. Hanford, "Automatic Generation of Test Cases," *IBM Systems Journal*, vol. 9, no. 4, 1970, pp. 242-257.

[69] M. Harman, "The Current State and Future of Search Based Software Engineering," in *FOSE*, 2007, pp. 342-357.

[70] M. J. Harrold, "Testing: A Roadmap", in *ICSE'00 Proceedings of the Conference on The Future of Software Engineering*, 2000, pp. 61–72.

[71] W. E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, Jul. 1977, pp. 266-278.

[72] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proceedings of the Winter 1992 USENIX Conference*, 1992, pp. 125-138.

[73] J. de Halleux and N. Tillmann, "Parameterized Unit Testing with Pex," in *TAP*, 2008, pp. 171-181.

[74] IEEE Standard, "1028–2008 —– IEEE Standard for Software Reviews and Audits," IEEE Computer Society, Aug. 2008.

[75] IEEE Standards Collection: Software Engineering, "IEEE Standard 610.12-1990," IEEE Computer Society, 1993.

[76] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A Concolic Whitebox Fuzzer for Java," in *NASA Formal Methods*, 2009, pp. 121-125.

[77] The Java PathFinder project, http://babelfish.arc.nasa.gov/trac/jpf.

[78] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, pp. 385–394, Jul. 1976.

[79] Klog, "The Frame Pointer Overwrite," *Phrack Magazine*, vol. 9, no. 55, Sep. 1999, Retrieved from: http://www.phrack.org/issues.html?issue=55&id=8#article.

[80] K. Ku, T. E. Hart, M. Chechik, and D. Lie, "A buffer overflow benchmark for software model checkers," in *ASE*, 2007, pp. 389–392.

[81] Y. Kim, M. Kim, and N. Dang, "Scalable distributed concolic testing: a case study on a flash storage platform," in *ICTAC*, 2010, pp. 199-213.

[82] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated Robustness Testing of Off-the-Shelf Software Components," in *FTCS*, 1998, pp. 230.

[83] C. Lattner and V. S. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–88.

[84] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux, "FloPSy: search-based floating point constraint solving for symbolic execution," in *ICTSS*, 2010, pp. 142-157.

[85] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural aliasing," in *PLDI*, 1992, pp. 235-248.

[86] G. J. Myers, "The Art of Software Testing," *3rd Edition, Wiley*, Nov. 2011.

[87] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification & Reliability*, vol. 14, pp. 105–156, Jun. 2004.

[88] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers, "Improving program slicing with dynamic points-to data," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, Nov. 2002, pp. 71-80.

[89] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS/ETAPS*, 2008, pp. 337-340.

[90] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of MacOS applications using random testing," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 1, 2007, pp. 46-54.

[91] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Communications of the ACM*, vol. 33, no. 12, Dec. 1990, pp. 32-44.

[92] P. McMinn and M. Holcombe, “Evolutionary testing using an extended chaining approach,” *Evolutionary Computation*, vol. 14, no. 1, Apr. 2006.

[93] D. Molnar, X. C. Li, and D. A. Wagner, “Dynamic test generation to find integer bugs in x86 binary Linux programs,” in *Proceedings of the 18th conference on USENIX security symposium*, 2009, pp. 67–82.

[94] K. K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, “Directed symbolic execution,” in *SAS*, 2011, pp. 95–111.

[95] R. Majumdar and K. Sen, “Hybrid Concolic Testing,” in *ICSE*, 2007, pp. 416-426.

[96] P. G. Neumann, “Forum On Risks To The Public In Computers And Related Systems,” Retrieved from http://catless.ncl.ac.uk/Risks.

[97] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, “CCured: type-safe retrofitting of legacy software,” *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 3, May 2005, pp. 477-526.

[98] National Institute of Standards & Technology, U.S Department of Commerce, “The Economic Impacts of Inadequate Infrastructure for Software Testing,” in *Planning Report 02-3*, May 2002.

[99] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs,” in *Proceedings of the 11th International Conference on Compiler Construction*, 2002, pp. 213-228.

[100] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in *PLDI*, 2007, pp. 89-100.

[101] National Vulnerability Database, http://nvd.nist.gov/.

[102] V. Okun, W. F. Guthrie, R. Gaucher, and P. E. Black, “Effect of static analysis tools on software security: preliminary investigation,” in *Proceedings of the 2007 ACM workshop on Quality of protection*, 2007, pp. 1-5.

[103] A. One, “Smashing the stack for fun and profit,” Retrieved from: http://insecure.org/stf/smashstack.html.

[104] J. Offutt and H. Hayes, “A semantic model of program faults,” in *ISSTA*, 1996, pp. 195–200.

[105] C. Pacheco, “Directed Random Testing,” *Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, (Cambridge, Massachusetts)*, Jun. 2009.

[106] The PaX Team, “Documentation for the PaX project,” Retrieved from: http://pax.grsecurity.net/docs/pax.txt.

[107] S. Person, “Differential Symbolic Execution,” *Ph.D. dissertation, University of Nebraska - Lincoln, Lincoln, NE, USA*, 2009.

[108] R. S. Pressman, “Software Engineering: A Practitioner's Approach”, *7th Edition, McGraw-Hill Science/Engineering/Math*, Jan. 2009.

[109] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, “Combining unit-level symbolic execution and system-level concrete execution for testing NASA software,” in *ISSTA*, 2008, pp. 15-26.

[110] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?” in *ISSTA*, 2011, pp. 199-209.

[111] C. S. Păsăreanu and W. Visser, “Verification of Java Programs Using Symbolic Execution and Invariant Generation,” in *SPIN*, 2004, pp. 164-181.

[112] C. S. Păsăreanu and W. Visser, “A survey of new trends in symbolic execution for software testing and analysis,” *STTT*, vol. 11, pp. 339–353, Oct. 2009.

[113] D. Qi, H. D. T. Nguyen, A. Roychoudhury, “Path exploration based on symbolic output,” in *SIGSOFT FSE*, 2011, pp. 278–288.

[114] X. Qu and B. Robinson, "A Case Study of Concolic Testing Tools and their Limitations," in *ESEM*, 2011, pp. 117-126.

[115] O. Ruwase and M. S. Lam, "A Practical Dynamic Buffer Overflow Detector," in *NDSS*, 2004.

[116] C. V. Ramamoorthy, Siu-Bun F Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, Dec. 1976, pp. 293-300.

[117] RTCA, Inc., "Document RTCA/DO-178B," *U.S. Department of Transportation, Federal Aviation Administration, Washington, D.C*, 1993.

[118] M. Ruse, T. Sarkar, and S. Basu, "Analysis & Detection of SQL Injection Vulnerabilities via Automatic Test Case Generation of Programs," in *SAINT*, 2010, pp. 31-37.

[119] K. Sen, "Scalable Automated Methods for Dynamic Program Analysis," *Ph.D. dissertation, University of Illinois at Urbana-Champaign*, 2006.

[120] K. Sen and G. Agha, "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools," in *CAV*, 2006, pp. 419-423.

[121] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[122] Symantec Corporation, "Internet Security Threat Report 2013 Volume 18," Retrieved from: http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf.

[123] R. A. Santelices and M. J. Harrold, "Exploiting program dependencies for scalable multiple-path symbolic execution," in *ISSTA*, 2010, pp. 195–206.

[124] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/SIGSOFT FSE*, 2005, pp. 263–272.

[125] The Telegraph, "Knight Capital 'has 48 hours' to save itself after IT glitch causes $440m loss," Aug. 2012, Retrieved from: http://www.telegraph.co.uk/finance/markets/9448893/Knight-Capital-has-48-hours-to-save-itself-after-IT-glitch-causes-440m-loss.html

[126] M. Staats and C. S. Păsăreanu, "Parallel symbolic execution for structural test generation," in *ISSTA*, 2010, pp. 183–194.

[127] A. Tomb, G. Brat, and W. Visser, "Variably interprocedural program analysis for runtime error detection," in *ISSTA*, 2007, pp. 97-107.

[128] N. Tillmann and J. de Halleux, "Pex – white box test generation for .NET," in *TAP*, 2008, pp. 134–153.

[129] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," in *OOPSLA*, 2011, pp. 189–206.

[130] W. Visser, C. S. Păsăreanu, and R. Pelánek, "Test input generation for java containers using state matching," in *ISSTA*, 2006, pp. 37-48.

[131] M. Weiser, "Program slicing," in *ICSE*, 1981, pp. 439–449.

[132] E. J. Weyuker, "The evaluation of program-based software test data adequacy criteria," *Communications of the ACM*, vol. 31, no. 6, Jun. 1988, pp. 668-675.

[133] J. A. Whittaker, "What Is Software Testing? And Why Is It So Hard?" *IEEE Software Journal*, vol. 17, no. 1, pp. 70–79, Jan. 2000.

[134] K. E. Wiegers, "Peer Reviews in Software: A Practical Guide," *1st Edition, Addison-Wesley*, Nov. 2001

[135] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *NDSS*, 2000.

[136] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for C programs," in *PLDI*, 1995, pp. 1-12.

[137] J. A. Whittaker and H. Thompson, "How to Break Software Security," *Addison Wesley*, May 2003.

[138] R. G. Xu, "Symbolic Execution Algorithms for Test Generation", *Ph.D. dissertation, University of California at Los Angeles Los Angeles, CA, USA*, 2009.

[139] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *FSE*, 2010, pp. 257-266.

[140] T. Xie, D. Marinov, W. Schulte, and David Notkin, "Symstra: a framework for generating object-oriented unit tests using symbolic execution," in *TACAS*, 2005, pp. 365-381.

[141] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness–guided path exploration in dynamic symbolic execution," in *DSN*, 2009, pp. 359–368.

[142] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux, "Precise identification of problems for structural test generation", in *ICSE*, 2011, pp. 611–620.

[143] YouTube, "Ariane 5 Explosion," Retrieved from: http://www.youtube.com/watch?v=gp_D8r-2hwk.

[144] H. Yun, "Non-linear Arithmetic Support for CREST," *Technical Report*, 2012.

[145] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, Dec. 1997, pp. 366-427.