TOWARDS THE AUTOMATION OF VULNERABILITY DETECTION IN SOURCE CODE

HAI ZHOU LING

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE & SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

DECEMBER 2009

Your file Votre référence
ISBN: 978-0-494-67245-7
Our file Notre référence
ISBN: 978-0-494-67245-7

Abstract

Towards the Automation of Vulnerability Detection in Source Code

Hai Zhou LING

Software vulnerability detection, which involves security property specification and verification, is essential in assuring the software security. However, the process of vulnerability detection is labor-intensive, time-consuming and error-prone if done manually. In this thesis, we present a hybrid approach, which utilizes the power of static and dynamic analysis for performing vulnerability detection in a systematic way. The key contributions of this thesis are threefold. First, a vulnerability detection framework, which supports security property specification, potential vulnerability detection, and dynamic verification, is proposed. Second, an investigation of test data generation for dynamic verification is conducted. Third, the concept of reducing security property verification to reachability is introduced.

Acknowledgments

First of all, I would like to thank Dr. Mourad Debbabi, my supervisor, for his continuous support and insightful guidance. He welcomed me to the Trusted Free and Open Source Software (TFOSS) research team in September 2007. It was a great pleasure for me to conduct this thesis under his supervision.

I would like to express my gratitude to the members of my evaluation committee: Dr. Terry Fancott and Dr. Thiruvengadam Radhakrishnan.

I gratefully acknowledge my fellow researchers in TFOSS group: Aiman Hanna, Zhenrong Yang, Xiaochun Yang, Syrine Tlili, Mourad Azzam, Amine Boukhetouta, Dima Al-Hadidi, and Nadia belblidia.

I would like to thank my family. Many thanks to them for their continuous support, understanding and love.

Finally, I would like to thank all of those who supported me in any respect during the completion of the thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

Free and Open Source Software (FOSS) are programs whose licenses give users the freedom to run the programs for any purpose, to study and modify the programs, and to redistribute copies of either the original or modified programs (without having to pay royalties to previous developers) [94]. Today, more and more effort and attention are devoted to the development of FOSS as the use of FOSS has risen to great prominence [30]. FOSS is being accepted as a viable alternative to commercial software. For example, since 2000, Netcraft has been separately counting "active" web sites. In the April 2009 web server survey conducted by Netcraft, among the 231,510,169 web sites, Apache, which is an open-source web server, remains in the lead, as it has since 1996, with a total of over 106 million sites, which counts for the 45.95% of the market share, followed by Microsoft-IIS with over

67 million [20]. For cost-efficiency reason, more and more organizations, including government and military, have started to consider the deployment of FOSS applications [99]. In 2006, Forrester Research suggested to North American and European enterprises that "firms should consider open source options for mission-critical applications" [55].

Software engineering methodologies have been relatively successful in producing large software effectively, but it has not been so successful in producing secure software immune to abuse and malicious attacks. This has been demonstrated recently by many headline news about computer security attacks, spreading viruses, e-mail spams, and economic loss due to these security breaches [92].

With the obvious growth and the importance of FOSS today, FOSS security has become a very challenging concern for both industry professional and academician. Due to different technologies and market factors, open-source software often carry various security vulnerabilities, some of which can be very severe if exploited. To mitigate the problem, a serious effort should be placed on open-source software vulnerability testing. The automation of such process is becoming a necessity, which will result in reducing development costs and also improving the quality of the software under test. Among the techniques for vulnerability detection, static analysis is powerful in detecting vulnerabilities. However, static analysis techniques usually generate some amount of false positives. To reduce the number of false positives, effort should also be made.

1.2 Objectives

In this thesis, the goal is to investigate the automation of open source software security vulnerability detection with minimized number of false positives. To achieve this goal, it is important to automate the process of test data generation, which is needed for dynamic verification. Accordingly, the objectives of this thesis are illustrated as follows:

- Investigate the use of static analysis approach to identify risky set of vulnerabilities.
- Elaborate techniques that will generate test data to dynamically assess vulnerabilities.

1.3 Contributions

This thesis makes the following contributions to the development of an automatic and extensible solution for vulnerability detection:

- A framework for vulnerability detection that encompasses static analysis, dynamic analysis, code instrumentation, and test data generation;
- An investigation of different approaches for test data generation. In Particularly, we investigate the use of push-down model checking with program slicing, and hybrid (static and dynamic) test data generation;
- An investigation of the reduction of security property verification to reachability.

The model-checking based static analysis technique used in this thesis is leveraged from the research [88, 98].

1.4 Organization of the Thesis

The thesis is organized as follows: in the following chapter, we briefly introduce traditional software testing, security testing, and test data generation. Chapter 3 describes our approach for vulnerability detection and the system architecture. Chapter 4 explains the concept of test data generation using Moped reachability checker with program slicing technique. Chapter 5 presents a hybrid approach for test data generation. The integrated system for test data generation and the experimental results are provided in Chapter 6. Chapter 7 concludes our research work.

Chapter 2

Security Testing

This chapter briefly introduces traditional software testing that focuses on software quality, security testing that concerns software security, and different techniques for test data generation.

2.1 Traditional Software Testing

Software testing is an empirical investigation conducted to provide stake holders with information about the quality of the product or service under test [68]. The main purpose of software testing is to evaluate an attribute or capability of a program or system and determine that it meets its required results [60]. Software testing is not only a very labor-intensive but also expensive process. The software testing cost can account for 50% of the total software development cost [78]. The traditional software testing usually includes correctness testing, performance testing, and reliability testing.

For the correctness testing, an oracle is needed to validate whether the code implementation follows intended design. The person that performs the test may or may not know the inside details of the software under test. Accordingly, the correctness testing can be classified into two groups: white-box testing and black-box testing [82]. For black-box testing, the testers do not have the inside view of the structure of the software. It is test data-driven. The correctness testing is conducted by feeding test data to the software under test and observe the output. Contrary to black-box testing, while-box testing requires access to the source code. It also requires the generation of test cases to achieve statement coverage, and branch coverage.

Performance is another important benchmark of software quality. The execution of software should be performed in a finite time or finite resource. Performance testing is used to determine the speed of a system under a particular workload. The performance of a software can be evaluated in the aspects, such as resource usage, throughput, and stimulus-response time [82].

Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment [31]. Software reliability testing is a process that conducts quantitative reliability assessments for the software under test in terms of robustness and stress.

Traditional software testing concerns on the software quality, such as correctness, efficiency, and stability. However, software that meets the traditional quality criteria might not be secure. In recent years, since more and more vulnerabilities have been reported, software security concern grows. To assess the software security, security testing is needed.

2.2 Security Testing

Security testing is the process to verify that critical software protects data and maintains functionality as intended. Due to the increase of software security concerns, security testing has now become more and more important. Traditional software testing mainly focuses on functional testing and quality assurance. Security testing is often fundamentally different from traditional testing because it emphasizes what an application should not do, rather than what it should do [49]. In this thesis, we mainly focus on the software security vulnerability detection.

2.2.1 Vulnerabilities

The symptoms of security vulnerabilities are very different from those of traditional bugs [95]. Security testing is primarily performed to verify a system's conformance to security requirements (security properties) and to identify potential security flaws within the system. Security vulnerabilities are the consequence of violating security properties. There are six basic types of security properties [96]:

1. Confidentiality: ensuring that information is accessible only to those authorized to have access;

2. Integrity: ensuring the information transmitted are not altered by others;

3. Authentication: verifying that someone is who they claim to be;

4. Authorization: granting or denying access to resources;

5. Availability: assuring information and services will be ready for use when needed;

6. Non-repudiation: preventing the later denial that an action or communication happened.

The primary reason for software security testing is to identify potential vulnerabilities, which violate the security properties. Security testing goes deeper than simple black-box probing on the presentation layer and even beyond the functional testing of security apparatus [74]. It relies on a combination of creativeness, expansive knowledge bases of best practices.

There is a huge demand for security vulnerabilities testing. Figure 1 shows that the number of software security vulnerabilities reported has increased a lot since 1995, according to the CERT Statistics [4]. Up to the end of the third quarter of 2008, the total number of software security vulnerabilities has reached 44,074.

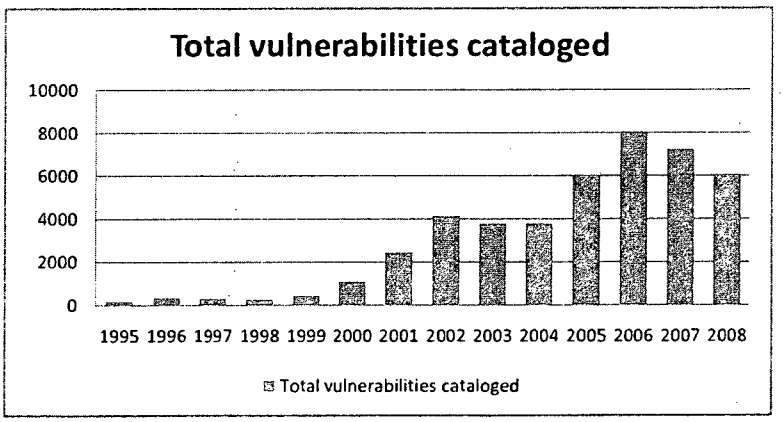


Figure 1: CERT Statistics: Cataloged Vulnerabilities

The cost could be huge once the vulnerabilities are exploited. For instance, in 2001, the Code Red and Code Red II viruses infected tens of thousands of systems running Microsoft Windows NT and Windows 2000 server software, causing an estimated 2 billion in damages [13]. In August 2007, the online job site Monster.com suffered a security breach that reportedly resulted in the theft of confidential information of about 1.6 million job seekers [34].

Vulnerabilities differ in a variety of formats. Some common types of vulnerabilities are illustrated as follows [2,21,90,97]:

- Memory management: this category contains vulnerabilities, such as buffer overflows, integer overflow, dangling pointers, memory leak, double free, use after free, free of unallocated memory, use of unchecked null returns, and use of uninitialized pointers. This class of vulnerabilities may result from erratic program behavior in memory manipulation.

- Input validation: this category contains vulnerabilities, such as format string bugs, assumption of null-termination, improperly handling shell metacharacters, SQL injection, code injection, E-mail injection, directory traversal, cross-site scripting, HTTP header injection, and HTTP response splitting. These vulnerabilities result from lacking validation of input parameters.

- Race conditions: this category contains vulnerabilities, such as time-of-check-to-time-of-use (TOCTOU) and symlink races, which result from creating files in an insecure way. This class of vulnerabilities are caused by the unexpected dependence

on the relative timing of events.

- API abuse: this category contains bugs like dangerous function, directory restriction error, and failure to follow guideline/specification. These vulnerabilities are caused by a caller failing to fulfill the API contract.

- Password management: this type of vulnerabilities results from insecure password manipulation. It contains vulnerabilities, such as empty string password, hard-coded password, and password plaintext storage.

Most of the vulnerabilities are caused by buggy code. Only about 36% of vulnerabilities result in configuration or design problems; the rest are due to programming errors [15]. For example, cross-site script insertion is due to lack of sanitization for data supplied in the HTTP Host header. Most of the buffer overflow vulnerabilities are caused by the weak or non-existent bounds checking on its input. Code injection vulnerabilities result from the mixing of code and data. This typically happens when attempting to dynamically generate commands, such as SQL to a database engine with user input [59]. Symlink vulnerabilities are due to the ambiguity as to which file is affected. This kind of vulnerabilities can be exploited by making a program affect a file that is not expected to be modified.

2.2.2 Vulnerability Detection Techniques

One of the important components of security testing is vulnerability detection. Various software vulnerability detection techniques exist. Here we list some of the most popular vulnerability detection techniques.

- Extended type systems: this technique addresses programming errors detection in the compilation process by extending the type system. Type safety check is an internal phase of some new programming languages, such as Java and C#. As a result, programs written in such languages are secure against to certain security vulnerabilities, such as illegal memory access. However, some programming languages, such as C and C++, are non-strong-typing ones [83]. Research efforts have been made to extend the existing type systems of these languages to incorporate safety check. For example, CCured [79] targets memory safety for C programs by extending C's type system through separating pointer types according to their usage [79, 99]. This technique is limited by its nature. To use this approach for vulnerability detection, it requires that the security properties is able to be expressed in terms of type inconsistencies.

- Metacompilation [58]: this is a static analysis technique that uses meta, a high-level program abstraction that is based on State Machine (SM), to specify security properties. It has two advantages: first, many security properties can be expressed in SM. Second, the abstraction is easy to understand since SM is a familiar concept in programming. The transitions between states are labeled as patterns, which match source code actions. Any transactions that lead to an error state indicate a violation of the security property. This technique usually focus on temporal properties. Coverity Prevent [5] is a static code analysis tool that implements metacompilation techniques. It focuses on C, C++, C# and Java source code.

- Model checking [41]: it is a technique for formal verification: a desired behavioral property of a reactive system is verified over a given system (the model) through exhaustive enumeration of all the states reachable by the system and the behaviors that traverse through them. MOPS [40] is a model checker that uses this technique. It focuses on detecting violations of temporal safety properties. The tool works in two phases. In the first phase, it transforms the program under test into a pushdown automaton [62]. In the second phase, the model checker intersects the pushdown automaton with finite state automaton, which defines the security property, to build a pushdown system. It then traverses the pushdown system in search of reachable error states. When such a reachable error state is detected, MOPS reports the security property violation with the offending path in source code. There is another model-checker called Moped [69], which utilizes a push-down system to simulate the execution of a program. Moped performs reachability analysis of a specified program point of a model, which is represented by Remopla language, and it can generate a concrete execution path for each reachable program point. The research [88, 98] elaborated the possibility and scalability of using Moped model checking for vulnerability detection.

- Pattern matching: this technique detects patterns in the parse tree after the code is analyzed. A parser scans the source code and constructs a parse tree. A rule-based system then tries to find patterns in the parse tree. The security concern is defined as the pattern. Any match of these patterns means a potential violation of the security property. PSCAN [23] is a tools that utilizes the pattern matching technique to detect

format string vulnerabilities in C code. This technique has limitations since it relies on the pattern to define the security properties. Only a few security properties can be specified as patterns. Most of the security properties are temporal ones, which comprise a sequence of program actions taken at a different time. For most temporal properties, they cannot be correctly specified as patterns.

- Program annotation induction: program annotation is the task of discovering a set of invariant assertions and documenting a program with them. These invariants describe the workings of the program by detailing relationships between the different variables at specific points [44]. Program annotation induction technique defines security properties as a set of preconditions and postconditions. The programs under test are annotated with these properties before verification. The induction is conducted as follows: for each function to be checked, the constraints that are defined as precondition are propagated, then the accumulated constraints are verified against the postconditions. During this process, any inconsistency detected is considered to be a violation of the security property in concern. Some vulnerability detection tools, such as Daikon [7] and Splint [26], incorporate this technique. This approach is limited by the fact that it is difficult to fully automate the annotation process, especially for complex software.

- Fuzz testing: it is a dynamic software testing technique that provides random data to the inputs of a program. The failure of the program indicates defects are detected. The advantage of fuzz testing is that the test design is simple, and it can be conducted

without preconceptions about system behavior [86]. This technique has been used in some vulnerability detection frameworks such as Peach Fuzzing Platform [22] and SPIKE [25]. Due to fact that it relies on random test data, this technique may suffer from poor code coverage.

- Fault injection [91]: this is a dynamic analysis technique that detects defects by observing system behavior in the presence of faults. This technique can be categorized into two types: compile-time injection and runtime injection [91]. Compile-time injection is done in the source code, where simulated faults are injected into the system. For runtime injection, a software trigger is used to inject a fault into a software system during its execution. Various faut injection techniques, such as Ferrari [67], and Holdeck [14], have been developed.

- Program monitoring: this is a dynamic analysis technique, which monitors the execution of the program under test. Insure++ [16] is an example of tools using this technique. It can automatically detect erroneous memory operations, such as accesses to freed memory, array bound violations, and freeing unallocated memory. Insure++ works as follows: first, it conducts code instrumentation for monitoring at the source code level. Second, it executes the instrumented code and monitor the execution to detect any erroneous memory operations.

A non-exhaustive list of existing leading security analysis tools [87,99] is shown in Table 1. These tools use various program analysis techniques to detect security vulnerabilities in software.

Many vulnerability detection tools are mostly specialized to certain vulnerabilities and have limited ability and generality. For example, Dmalloc is well-known in detecting memory management related vulnerabilities, such as buffer over-flow. BOON is specialized at applying integer range analysis to check if there is out-of-boundary array operation in a C program. ITS4 focuses on dangerous function calls.

Static analysis tools are usually successful in detecting vulnerabilities. However, they produces some false positives. On the other hand, dynamic analysis tools are precise, but they have run-time overhead and need full path coverage test cases. This prevents them from conducting exhaustive detection along all the paths as static analysis tools do.

2.3 Test Data Generation

Test data generation is a process of searching program inputs that satisfy selected testing criteria to make the execution of a program under test follow a specific path, or to reach a program point. In many cases, this is essential, especially for dynamic analysis, where such behavior is desired.

Generating test data in order to traverse a path involves solving a system of equations. If the system has no solution we can conclude that the path given is indeed infeasible. However, solving an arbitrary system of equations is undecidable [46]. Although test data generation is an undecidable problem, researchers still attempt to develop various methods and have made some progress.

Tool's Name	Description
BOON [1]	A static security testing tool detecting buffer overflow in C code
Coverity Prevent [5]	Static analysis tool for general security vulnerability detection in programs written in C/C++/Java/C#
CodeWizard [3]	Static analysis tool for finding dangerous code constructs in programs written in C/C++
Cppcheck [6]	Security testing tool for statically checking C/C++ programs for security vulnerabilities, such as memory leaks, and coding mistakes
Dmalloc [8]	Dynamic memory management flaws detector for C applications
EXE [9]	General-purpose dynamic code analysis tool for C applications
Insure++ [16]	Runtime memory errors detector for C/C++ code
ITS4 [17]	Static analysis-based tool that finds defects like dangerous function calls and potential buffer overflows in C/C++ applications
JNuke [32]	Hybrid framework for verification and model checking of Java programs, which combines run-time verification, explicit-state model checking, and counter-example exploration
Klocwork [18]	General-purpose static code analysis framework targets applications written in C/C++/Java
MC Checker [19]	Model checker for verifying temporal safety properties in C code
MOPS [40]	Model checker for detecting violation of temporal safety properties
Ounce [71]	Static security testing tools detecting vulnerabilities in source code based on model analysis
PC-lint [11]	General-purpose static code analysis tool for programs written in C/C++
Rational PurifyPlus [24]	A framework used by software developers to detect memory access errors in programs, especially those written in C or C++
RATS [66]	Static security auditing utility for C and C++ code that finds potentially dangerous function calls
Splint [26]	Static analysis tool targeting buffer overflow, format errors for C code
Valgrind [28]	An instrumentation framework for building dynamic analysis tools for detecting memory management for programs written in C/C++/Java/Perl/Python/Fortran/Ada

Table 1: Security Analysis Tools

The past few years have witnessed a great research interest targeting automatic test data generation. Many approaches have been proposed including random test data generation [37], directed random test data generation [54], genetic and evolutionary algorithms [39], chaining approach [48], iterative relaxation approach [56], and symbolic execution for test data generation [36, 38, 42, 52, 63, 84]. Recently, model-checkers have been suggested for test-case generation [50].

In 1996 Ferguson and Korel [48] classified these methods into three categories:

- Random approach: randomly chooses an arbitrary input value from the set of all possible input values of the program under test;
- Goal-oriented approach: generates test data for a program point regardless which path to follow;
- Path-oriented approach: generates test data for a specific path.

Another type of classification of test data generation approaches is based on the software analysis technique they use. It categories the approaches as follows:

- Static approach: does not require the program under test to be executed, but work on the analysis of the program under test;
- Dynamic approach: involves the repeated execution of the program under test during a directed search for test data that meet the desired criterion.

Static techniques typically use symbolic execution to obtain constraints on input variables for a particular testing criterion. Symbolic execution works by traversing a control

flow graph of the program under test and constructing symbolic representations of the internal variables in terms of the input variables. Branches within the code introduce constraints on the variables. Solutions to these constraints represent the desired test data. There are limitations with this approach. It is difficult to use symbolic execution for various cases including recursion, array operations, where indices depend on input data, and loop structure [89]. In 1976, Miller [77] proposed a dynamic approach to test data generation, in which test data generation is formulated and solved as a numerical optimization problem.

Table 2 categories most of the approaches to test data generation based on the classifications mentioned above.

Test Data Generation Techniques	Description
Chaining approach [48]	Goal-oriented & dynamic
Function-minimization [70]	Path-oriented & dynamic
Genetic algorithms [76]	Path-oriented & dynamic
Iterative relaxation [56]	Path-oriented & dynamic
Model-checkers [40, 69]	Goal-oriented & static
Random test data generation	Random & dynamic
Symbolic execution [42]	Path-oriented & static

Table 2: Test Data Generation Approaches

The chaining approach was proposed by Roger Ferguson and Bogdan Korel [48] in 1996. In the chaining approach, test data are derived based on the actual execution of the program under test. The chaining approach uses data dependence analysis to guide the search process, i.e., data dependence analysis automatically identifies statements that affect the execution of a particular selected statement. The chaining approach uses these statements to form a sequence of statements that is to be executed prior to the execution of the selected statement. The basic idea of chaining approach is to build up a chain of nodes

that are relevant to the execution of the goal node iteratively during execution.

Inspired by Darwin's evolution theory, Holland proposed the genetic algorithms in 1992 [61], which is a heuristic search algorithm based on the idea of maintaining a population of sample solutions, each of which fitness has been calculated. The successive populations are evolved using the genetic operations, such as crossover and mutation. The basic idea of genetic algorithms is that through the use of the genetic operations the population will converge towards a global solution [61]. The concept of applying genetic algorithms for test data generation was introduced by Michael et al in 1997 [76]. Given a set of sample data (population of sample), test data is generated through the evolution of manipulating the sample data.

Iterative relaxation for test data generation was proposed by Gupta, Mathur, and Soffa in 1998 [56]. In this technique, the input value is arbitrarily chosen from a given domain during the first iteration. In the following iterations, the input is iteratively adjusted to obtain desired outcome so that all the branch predicates along a given path are satisfied. During the iterations, the program statements that are relevant to the evaluation of each branch predicate on the given path are executed to construct a set of linear constraints. The constraints are then solved to obtain the adjustment for the input. The adjusted input then is used for the next iteration. This approach is dynamic and path-oriented.

Random testing is the simplest method of generation techniques and at the same time it is the most unreliable technique for testing [46]. This technique has the weakest coverage criteria. It randomly picks any of the paths from Control Flow Graph (CFG) and generate test data for that path. The obvious disadvantage of random approach is the low efficiency

since it merely relies on probability. Random testing approach is often used as a benchmark since it has the lowest efficiency in test data generation.

There are some test data generators that make use of the symbolic execution technique [36, 38, 42, 52, 63]. In symbolic execution, instead of using actual values, symbols representing arbitrary values is used. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. The basic idea of symbolic execution is to generate an expression in terms of input variables [46]. Symbolic execution has been popular since the 70's. However, it has some drawbacks: first it is expensive in consuming resources. Second, it has technical problems in handling input variable dependent loops, subprogram calls, and array references [43].

In 1990, Korel proposed an alternative approach to test-data generation based on function-minimization methods and dynamic analysis [70]. When the program under test is executed, its execution flow is monitored. Any execution flow that leads the execution of the program under test away from the selected path is classified as undesirable one. During the program execution, if an undesirable execution flow is detected, the function-minimization search is conducted to find the values of input variables for which the chosen path is traversed. Furthermore, dynamic data-flow analysis is utilized to determine those input variables responsible for the undesirable program behavior, which dramatically increases the efficiency of the search process [70].

A model-checker is a tool that is designed for formal verification. It takes two inputs: the model of the program under test and a temporal logic formula, which describes the security property in concern. It explored the entire state space of the model to verify if

there exist any violations of the security property. If violations are detected, then the model-checker returns a trace which is a sequence of states leading from the initial state to error state. Each state in the trace describes the values of all variables, including input variables, of the model. The values of input variables are used as test data [50].

These approaches vary in nature and target different goals. For instance, random test data generation strives to feed the test data to the program under test randomly in an attempt to execute different paths and hopefully result in some useful test suites being created. This approach is easy to implement. However, it is time-consuming in test data generation. Goal-oriented approach, on the other hand, targets a specific point in the software and attempts to generate test suites to reach this desired target without caring for which path to follow. In this thesis, the vulnerability detection framework we are going to propose is mainly targeting temporal security property, in which a sequence of statements need to be executed orderly to result in a violation. As such, we are interested in generating test data for a specific path, which might violate security properties. Consequently, path-oriented approach is more suitable to achieve reachability. On the other hand, symbolic execution suffers when generating test data for programs that contain dependent loops, subprogram calls, and array references. Due to this fact, it is not suitable to use symbolic approach to generate test data for complex programs.

2.4 Summary

To summarize, in this chapter we discussed the traditional software testing, security testing, and test data generation. We discussed different types of software testing, such as correctness testing, performance testing, and reliability testing. For security testing, we mainly discussed the types of vulnerabilities, and the vulnerability detection techniques. In test data generation part, we focused on the test data generation techniques.

Chapter 3

Approach

This chapter introduces our proposed approach for vulnerability detection. It focuses on security property specification, vulnerability detection and dynamic verification with test data generation.

3.1 A Synergy between Static and Dynamic Analysis

There are many different types of software analysis techniques for vulnerability detection. Among these, there are two major types: static analysis and dynamic analysis.

Static analysis is the process of evaluating software or a software component based on the syntactic and semantic information inferred from its form, structure, content and/or documentation without program execution [29]. Typically, static analysis is conservative and sound [47].

Some static analysis approaches like PSCAN [23] use pattern matching techniques to

look for insecure patterns, such as insecure function calls in the source code. This approach is simple, and effective in finding certain flaws. However, it is limited by nature since only few security properties can be specified and checked with string matching. In order to deal with more complex security flaws, some static analysis approaches utilize lexical analysis. It preprocess and converts the source files into a sequence of tokens and then match the resulting token stream against the predefined security specifications. However, most security vulnerabilities are not straightforward. To identify them, semantic interpretation is needed. To incorporate semantic analysis, Abstract Syntax Tree (AST), which is a structure describing the code in an abstract form is created. Static analysis is used to conduct a semantic analysis on the AST. It is conservative by considering all (syntactic) program paths. However, it might produce some amount of false positives.

There are two forms of static analysis for vulnerability detection: security code inspection and automatic static analysis. Security code inspection is time-consuming and requires the auditor to be a security expert. In addition, for large and complex programs, security code inspections may be impractical, and error prone. Automatic static analysis has advantages over manual security code inspection. First, the evaluation of a program can be done efficiently compared with security code inspection. Second, it does not require the operators to have the same level of knowledge in security as a human auditor. Consequently, using static analysis tools for vulnerability detection can reduce the cost and improve efficiency.

In contrast to static analysis, dynamic analysis derives properties that hold for one or

more execution by examining the running program usually through program instrumentation [72]. Dynamic analysis is precise. It usually involves instrumenting the program under test to verify or record certain aspects of its runtime behavior. The instrumented code is used to collect the precise information needed to address a particular property. In addition, with code instrumentation, dynamic analysis is able to relate changes in program inputs to changes in internal program behavior and program outputs, since all are directly observable and linked by the program execution [35].

While dynamic analysis has its advantages, it is not comprehensive since it suffers from the coverage issue. To achieve the latter, we need to observe every possible execution, which might require an infinite number of executions. Due to this fact, dynamic analysis generally does not prove that a program satisfies a particular property. However, it can detect violations of properties as well as provide useful information to the security analyst about the behavior of the programs under test [35]. Therefore, dynamic analysis is not enough by itself to address the problem of vulnerability detection in general, but it is very suitable for vulnerability verification.

	Static Analysis	Dynamic Analysis
Pros	No runtime overhead Sound No test case design needed Analyzed program need not be complete	Runtime information available Minimize false positives Precise Check certain types of security vulnerabilities (e.g. memory management and pointer arithmetics) more efficiently
Cons	Can only theorize potential vulnerabilities False positives Require source code	Run-time overhead False negatives Require test cases Not comprehensive

Table 3: Static Analysis vs. Dynamic Analysis

Clearly, both types of analysis techniques have their tradeoffs. Table 3 summarizes the pros and cons of these two approaches. The two approaches are complementary in many aspects. Therefore, it is desirable to generate information from static analysis for runtime verification. Accordingly, if we combine static analysis and dynamic analysis, then we will establish a synergy that will result into an improved analysis. To this end, we propose a synergy bewteen static analysis and dynamic analysis for vulnerability detection. The static analysis will be accommodated to guide the dynamic analysis. Our approach uses static analysis to point to potential vulnerabilities, then dynamic analysis to verify whether they are actual vulnerabilities. To link the static analysis with dynamic analysis, test cases are needed. This is where the test data generation comes into play.

3.2 Description

To detect vulnerabilities in the systematic way, four factors need to be considered. In the first place, we need to be able to specify the security property. In the second place, we should be able to point to potential vulnerable program points/paths. In the third place, we should be able to generate test data for further dynamic analysis. In the fourth place, we should be able to conduct dynamic analysis to verify vulnerabilities. Accordingly, Figure 2 illustrates the components of our approach as well as the techniques used in each component.

The first component is the security specification, which allows a user to specify a security property by using security automata. The second component is potential vulnerability detection, which utilizes static analysis to detect potential vulnerabilities. The third component is test data generation, which uses static analysis, dynamic analysis, code instrumentation, Moped reachability checker, and program slicing to generate test cases for further vulnerability verification. The fourth component is dynamic vulnerability verification, which performs dynamic analysis to verify a vulnerability using the test cases that are generated. It utilizes techniques like security automata, and code instrumentation for program monitoring. The details of our approach will be explained in the following sections.

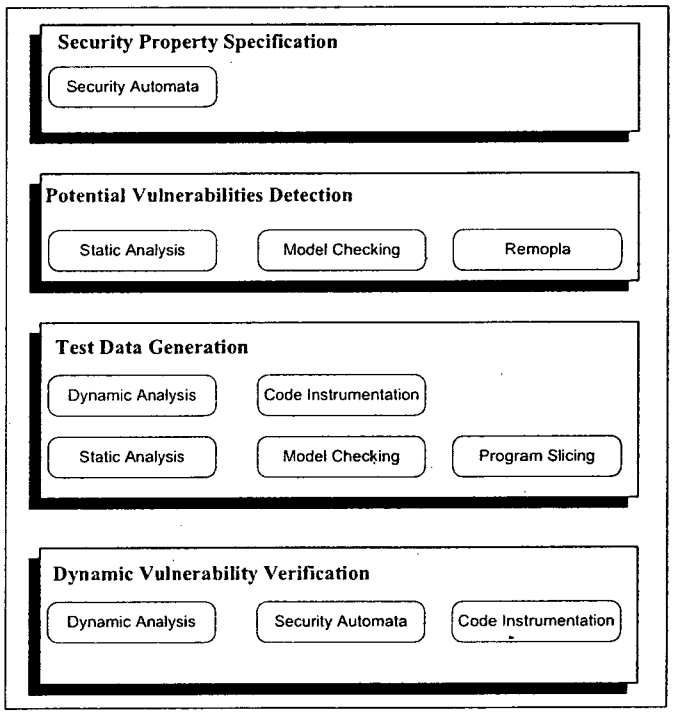


Figure 2: System Architecture

3.3 Security Properties Specification

To validate software against specific security violations, security analysts need to formally specify the security properties they need to test the software against. In this thesis, we use security automata [85] as the specification language for temporal security properties.

A security automaton is a deterministic finite or countably infinite state machine [73]. A temporal security property can be expressed by using a security automaton. Nodes inside a security automaton define the states of a program under test, and labeled transitions constitute program operations. The start node represents the initial state of the program under test. The error state is the violation of the security property in concern. The transitions between states depend on (i) the automaton state, (ii) the behavior of the program, and (iii) the argument of the method [64]. Figure 3 illustrates how a security property can be expressed as a security automaton. The security property specified in Figure 3 is related to privilege escalation: Since a privileged process has full access permission to the system, it should not make certain system calls that run untrusted programs without first dropping all privileges; Otherwise they may have full access to the system. One such system call is `execl`. To limit the privilege of a program executed by `execl`, a privileged process should drop privilege before calling `execl` [40]; failing to do this is a security vulnerability.

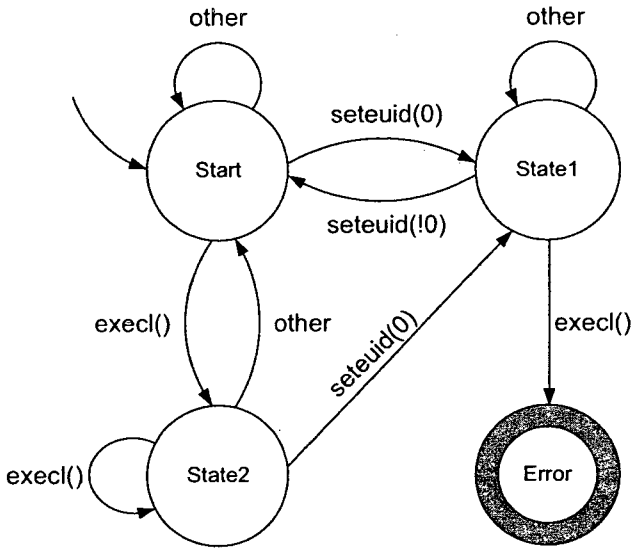


Figure 3: Sample Security Automaton

3.4 Code Instrumentation

To monitor the runtime behavior of a program under test and collect dynamic information, monitoring code needs to be added to the program. To achieve this goal, code instrumentation is necessary.

Various code instrumentation techniques exist. The main code instrumentation techniques include preprocessor macros instrumentation, library inclusion instrumentation, library replacement instrumentation, parser-level instrumentation, binary wrapping of object code, aspect-oriented programming (AOP) instrumentation, and compiler-assisted instrumentation [99].

We chose the compiler-assisted instrumentation approach. This choice is justified by the fact that compilers have the knowledge of the lexical structure and semantics of the

program being instrumented, and they are capable of building and analyzing the AST and the control flow graph of the program. This allows us to precisely select the exact program points where instrumentation code should be injected.

3.4.1 GCC

GNU Compiler Collection (GCC) [12] is chosen as the code instrumentation utility based on the following reasons:

- GCC is the default compiler for most open-source software;
- GCC is a multi-platform compiler. It is widely supported by various operation systems, such as Windows, Linux, Unix, etc;
- GCC can be extended because of the availability of the source code;
- GCC is a collection of compilers, which support many programming languages. Currently, the main GCC distribution contains front ends for C (gcc), C++ (g++), Objective C, Fortran, Java (GCJ), and Ada (GNAT);
- The Tree Single Static Assignment (TREE-SSA) [80] framework, which performs optimization for tree representation of source code, is merged into the official release of GCC (Starting from version 4.0). It offers an easy access to control-flow graphs, use-def chains, and point-to alias information that enable dataflow analysis;
- GIMPLE is the language-independent intermediate representation of GCC. With GCC's internal APIs, it is easy to manipulate the GIMPLE representation. Code

instrumentation done in this representation can be extended easily to support multi programming languages.

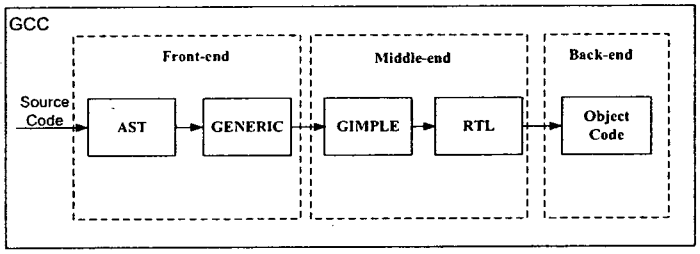


Figure 4: GCC Architecture

The GCC version 4 and above is divided into front-end, middle-end, and back-end. Different front ends exist for preprocessing different languages. The middle-end conducts transformation and optimization. The back-end, which is language-independent, is responsible to generate the final output: object code. This structure makes GCC to be capable of supporting a variety of languages. Given the source code, the front-end generates GENERIC representation, which is universal for all all languages. However, the GENERIC language is still difficult for the computer to understand since it contains complex statements and nested expressions. Accordingly, it is then transformed to a three-address code called GIMPLE in the middle-end. Temporary variables are used to simplify the the nested expression. Complicated structures are decomposed by using `goto` statements and corresponding labels. The middle-end further transforms the GIMPLE representation to Register Transfer Language (RTL). At the same time, the middle-end goes through a list of high-level optimization passes, such as alias analysis and dead code elimination. Consecutively,

the back-end generates object code. Figure 4 shows the relationship of the three components.

GENERIC and GIMPLE Languages

GENERIC is the language-independent intermediate tree structure that various GCC front-ends, which exist for different programming languages, produce. The GENERIC representation is a high-level AST representation. It is capable of representing all languages supported by the GCC compiler. The GCC middle-end performs many passes of AST-level analysis on the GENERIC tree. GENERIC representation is very well suited to represent the output of a parser. However, it is difficult to perform optimization on GENERIC due to its complexity.

In the GNU Compiler Collection, GIMPLE is an intermediate representation (IR) of the program, in which complex expressions are split into a three-address code using temporary variables [12]. Most CFG-level optimization passes are performed on the GIMPLE tree. Each GIMPLE tree is used to build a GIMPLE/CFG. Nested expressions are decomposed to a three-address form with temporary variables storing intermediate values. Control structures such as `for` and `while` loops in C are replaced by `gotos`. Complex conditional expressions are decomposed. For example, `if (cond1 || cond2) stmt;` is translated into

```
if (cond1) goto L1;
if (cond2) goto L1;
else goto L2;
```

```
L1:

stmt;

L2:
```

GIMPLE has internal representation for expressions. Table 4, extracted from GCC Internals [51], lists some of the expression nodes.

Expression Node	Description
INTEGER_CST	integer constants
VAR_DECL	variables
INDIRECT_REF	the object pointed to by a pointer
PLUS_EXPR MINUS_EXPR MULT_EXPR RDIV_EXPR	binary arithmetic operations
LT_EXPR LE_EXPR GT_EXPR GE_EXPR EQ_EXPR NE_EXPR	comparison operators
MODIFY_EXPR	assignment
COND_EXPR	conditional expressions
CALL_EXPR	calls to functions

Table 4: GIMPLE Internal Representation for Expressions

After version 4, GCC internals are very different from previous distributions. All code is translated into GIMPLE and analysis is conducted over these representations. Appendix A lists the GIMPLE representation of a sample C program.

3.4.2 GCC Extension

We extend GCC for code instrumentation by adding a new pass to it. The implementation is based on the GCC core distribution version 4.2.0 [10]. We keep all the functionalities

of the original GCC compiler. The instrumentation is done by modifying the GIMPLE representation in the middle-end during the compilation process. The implementation of the APIs for monitoring is written in C++ and built into a shared library with C interfaces. Figure 5 shows the work-flow of the GCC extension.

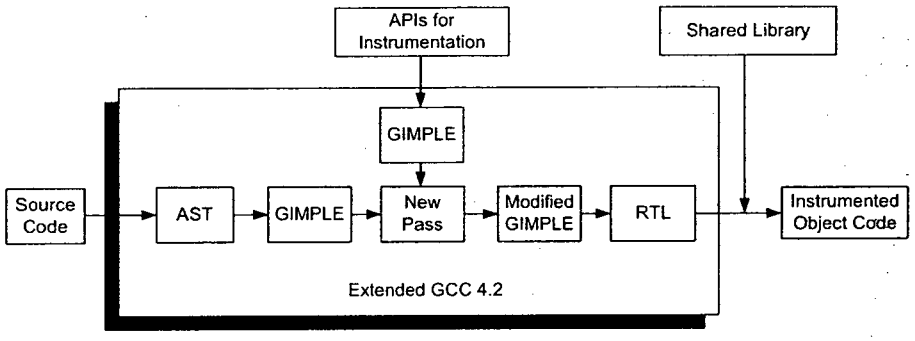


Figure 5: GCC Extension

Four steps are involved to extend GCC for code instrumentation:

1. Add a new pass to GCC;
2. Identify the locations to instrument;
3. Construct new GIMPLE representation of the calls of the APIs for monitoring;
4. Merge the original GIMPLE representation of the program under test with the new GIMPLE representation.

First, a new pass needs to be added to GCC, which divides its different level of tasks into passes. Yang [99] explains the details of how to add a new pass into GCC. All the passes are managed by the pass manager, which is located in the following GCC files:

passes.c, tree-optimize.c and tree-pass.h. The job of the pass manager is to run all the individual passes in the correct order, and do the standard bookkeeping that applies to every pass. Each pass is defined by a structure that represents everything we need to know about that pass, such as when it should be run, how it should be run, what intermediate language form or on-the-side data structures it needs, etc. The passes can be classified as follows:

- Initializers: such as pass_referenced_vars, pass_build_cfg
- Analysis: such as pass_build_ssa
- Optimizations: such as pass_loop2

```
struct tree_opt_pass                                    1
{                                                       2
const char *name;                                       3
bool (*gate) (void);                                    4
unsigned int (*execute) (void);                         5
struct tree_opt_pass *sub;                              6
struct tree_opt_pass *next;                             7
int static_pass_number;                                 8
unsigned int tv_id;                                     9
unsigned int properties_required;                       10
unsigned int properties_provided;                       11
unsigned int properties_destroyed;                      12
unsigned int todo_flags_start;                          13
unsigned int todo_flags_finish;                         14
char letter;                                            15
};                                                      16
```

Figure 6: Pass Description

Figure 6 shows the struct used to represent a pass. The main attributes of the struct are described as follows: name defines the name of the pass. The gate() function guards the entrance of the pass. If the gate() returns true, then the pass entry

point function `execute()`, which holds the code to run, is called; otherwise it will not be executed. The pointer `sub` points to the first one of a list of sub-passes to run, dependent on gate predicate. The pointer `next` points to sibling class passes are chained together with `NEXT_PASS` in `init_optimization_passes`. The `static_pass_number` is used as a fragment of the dump file name. Each pass can define its own separate timer `tv_id`. Timers are started/stopped automatically by the pass manager. Timer handles (`timevars`) are defined in `timevar.def`. The `todo_flags_start` and `todo_flags_finish` are flags indicating common sets things to do before and after a pass.

After we define a pass, the pass must be registered so that it can run in some particular order, and the pass manager arranges for everything to happen in the correct order. To register a pass, we modify the `passes.c`, which holds the pass manager. Figure 7 shows how to register our new pass `pass_security_tesing` in the pass manager, which is located in `passes.c`.

```
init_optimization_passes (void) {                    1
        ...                                          2
        NEXT_PASS(pass_mudflap_2);                   3
        NEXT_PASS (pass_security_tesing);            4
        NEXT_PASS(pass_free_datastructures);         5
        ...                                          6
}                                                    7
```

Figure 7: Pass Registration

Second, we need to identify the locations, in which we are interested to instrument. In this thesis, our points of interest are the following:

1. Beginning of the program

2. Beginning of each basic block, which is a straight-line sequence of code with only

one entry point and only one exit [12].

3. Environmental interactions [75]:

   (a) User interactions

   (b) Interactions with the file system

   (c) Interactions with memory

   (d) Interactions with the operating system via system calls

   (e) Interactions with other components of the same software system

   (f) Interactions with dynamically linked libraries

   (g) Interaction with other software via APIs

   (h) Interaction with other software via interprocess communication

   (i) Interactions with global data such as environment variables and the registry

   (j) Interactions with the network

   (k) Interactions with physical devices

   (l) Dependencies on the initial environment

4. Conditional statements

5. End of the program

To identify the locations to instrument, we traverse the GIMPLE tree and check the node type. For example, the conditional statements are expressed as a type of `COND_EXPR` in

GIMPLE. If the tree node of current iteration is of type COND_EXPR, then we find one of our interests.

Third, construct the GIMPLE tree for the function calls. For each type of C statement there is a function that constructs a tree node of the corresponding type. For example, `tree build_function_call(tree function, tree params)` builds a CALL_EXPR node for a function call. It takes the identifier of the function name and the arguments as its parameters. The function finds the function declaration using `lookup_name()` and type casts the arguments using `default_conversion()` [51].

Fourth, modify the GIMPLE tree, which represents the original program. After constructing the GIMPLE representation of the functions calls, we merge the new GIMPLE tree with the original one. Mostly, the code instrumentation is dealing with the GIMPLE tree. Table 5 shows the APIs provided by GCC that allow one to traverse a statement list and to manipulate it:

Function	Purpose
`tsi_start(stmt_list)`	get an iterator pointing at list head
`tsi_last(stmt_list)`	get an iterator pointing at list tail
`tsi_end_p(iter)`	is end of list?
`tsi_stmt(iter)`	get current element
`tsi_split_statement_list_before(&iter)`	split elements at iter
`tsi_link_after(&iter, stmt, mode)`	link chain after iter
`tsi_next(&iter)`	next element of the list
`append_to_statement_list(tree,&stmt_list)`	append tree to `stmt_list`

Table 5: Tree Construction API [51]

With GCC extension, we can instrument additional code into the program under test

for test data generation purpose. The code instrumentation is done during the compilation. Once the program under test is compiled successfully, an instrumented executable is generated.

3.5 Is Reachability Enough?

Assume we are able to point to potential security vulnerabilities using static analysis, and generate test data to reach the program sites in question. Can we say that we prove the existence of the security vulnerabilities? To answer this question, we need to know the types of security vulnerabilities that we are detecting.

There is a wide spectrum of software security vulnerabilities. Different classifications of those vulnerabilities can however be made based on their nature, the way they can be manifested, their level of harmfulness, the difficulty level to detect them, etc. In our research, we classify the vulnerabilities based on the matter of reachability, where the execution is sufficient to violate a security property. In that context, we refer to those vulnerabilities as reachability security vulnerabilities. Otherwise, we categorize them as non-reachability ones.

To detect non-reachability vulnerabilities, sometimes only generating test data to follow the suspicious path is not sufficient. In some cases, even we reach the target following a specific path, the violation of security properties might not be triggered. Examples of such include divide-by-zero errors, buffer-overflow, numeric underflow and overflow. This kind of vulnerabilities will be triggered only when some additional constraints are satisfied.

For example, the string copy library functions in C are vulnerable to buffer overflow attacks. The destination buffer must be large enough to hold the source string. This gives the attacker the opportunity to send an input that is larger than the buffer size, which will result into overflowing the buffer. For this vulnerability detection, the generation of test data that satisfies all constraints along a path to reach program points where the vulnerable functions are located is insufficient. To prove that a site is vulnerable, an additional constraint `sizeof(destination)`$\leq$`sizeof(source)` must also be satisfied. For instance, in Figure 8, only generating test cases to reach the vulnerability site `strcpy(dest, src)` is not enough. For example, if we generate test cases with x=13, y=0, the execution of the program can reach the vulnerable site. However, the buffer-over will not be triggered. To mitigate this problem, we add a new constraint `If(strlen(src)` $\geq$ `strlen(dest))` before the call of the `strcpy` function so that the test cased generated will trigger the buffer-overflow.

```
#include <stdio.h>
#include <stdlib.h>
int x,y;
scanf("%d",&x);
scanf("%d",&y);
char * dest = (char *) malloc(y + 16);
char * src = "buffer overflow";
If (x > 12){
   ...
   //vulnerable site
   strcpy(dest, src);
   ...
}else{
  ...
}
```

Figure 8: Is Reachability Enough?

Non-reachability Vulnerability	Additional Constraint
DIVIDE-BY-ZERO	divisor = 0
MEMCOPY [2]	size of destination$\leq$size of source
STRCPY [2]	size of destination$\leq$size of source
GETS [2]	size of input string$\geq$size of buffer
SCANF [2]	size of input string$\geq$size of buffer

Table 6: Non-reachability Vulnerability

By adding such a constraint into the constraint set along a suspicious path, we reduce the non-reachability problem into a reachability one. Table 6 shows some non-reachability vulnerabilities and the additional constraints which ensure the violation of the security properties.

Many of the non-reachability vulnerabilities are buffer-overflow related vulnerabilities. The United States Department of Homeland Security [2] reports a taxonomy that is comprised of 174 security vulnerabilities on its web site `buildsecurityin.us-cert.gov`. We look at the vulnerabilities listed one by one, and found out that among these 174, 71 vulnerabilities were buffer-overflow related vulnerabilities, which can further be reduced to reachability ones.

Based on our classification of the security vulnerabilities, and the introduction of adding additional constraints, which are used to guarantee the violation of the security property, we can verify the existence of a non-reachability vulnerability by generating test data that satisfies the original constrains along the suspicious path and the additional constrains. As a result, some non-reachability vulnerabilities could be converted into reachability ones.

3.6 System Design

The system is designed to use the synergy between static and dynamic analysis for vulnerability detection. It contains three main components. First, a static analysis tool [57, 98], namely the Static Vulnerability Revealer (SVR), is used to detect the vulnerabilities against the security properties defined by users. Second, a test data generator is responsible to generate test cases to provide the reachability to the vulnerable sites. Third, given the test data, a dynamic analysis tool named Dynamic Vulnerability Detection (DVD) [99] is utilized to verify the vulnerabilities detected by SVR so as to minimize false positives. In this thesis, we mainly focus on the test data generator. For test data generation, we utilize two different techniques. For small size software, such as embedded software, we use Moped reachability checker with program slicing to generate test data. Otherwise, a hybrid (combination of static and dynamic analysis) approach is utilized. The whole system design is illustrated in Figure 9.

The system starts with SVR, which takes the security specification and uses model checking techniques to detect suspicious paths that might contain the violations against the predefined security properties. Due to the nature of static analysis, false positives might also be generated. To verify a vulnerability, test data generation is needed. The test data generator takes the suspicious path, and tries to generate the test case such that the execution of the program can follow the suspicious path. Once the test data is generated, DVD takes the generated test data, executes the program, and monitors the execution of the program to verify the vulnerabilities.

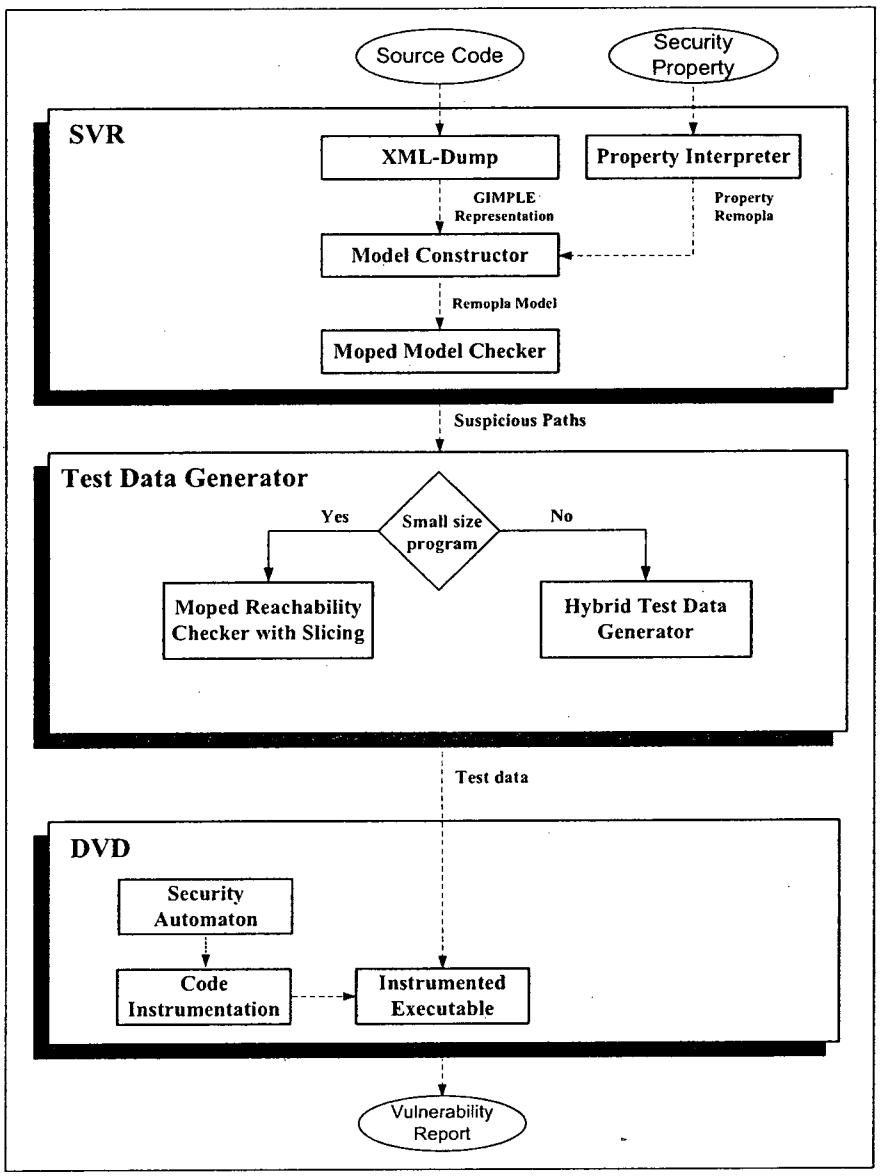


Figure 9: System Design

3.6.1 Static Vulnerability Revealer

Among the static vulnerability detection techniques, model-checking is a promising one. The research [88, 98] elaborated the possibility and scalability of using Moped model checker to detect vulnerabilities. SVR is based on that research.

Given source code of the program under test, and a formally specified security property, SVR main concern is to find out all (the largest set possible) program paths that have the potential of violating the security property in concern. We refer to these paths as suspicious paths. SVR is based on two major techniques: static analysis and model checking.

In SVR, a security property is modeled as a security automaton, specifying the erroneous behavior using sequences of program actions. An `error` state is introduced to represent the risk state for each automaton. A synchronization mechanism allows program actions to trigger the state changing of the automaton. If the `error` state is reached, a sequence of program actions violating the given property has been detected.

SVR automates the model construction process, including the generation of program model, property model, and the synchronization mechanism. Also, to mitigate the state space explosion problem of model checking, SVR uses static analysis to reduce the state space. By analyzing the CFG and Data Flow Graph (DFG), SVR is able to extract property-relevant behaviors and discard property-irrelevant information. The model generated by the static analysis is represented by a model specification language Remopla.

SVR employs a model-checker called Moped [69]. Moped performs reachability analysis of a specified program point of the given Remopla program, and it can generate a concrete execution path for each reachable program point. A set of program points that may

trigger the automaton to reach the `error` state is extracted at the static analysis phase, each of which is given to the model-checker for reachability analysis. In other word, SVR transforms the vulnerability detection problem into reachability analysis. More details about SVR can be found in [98].

3.6.2 Test Data Generator

To verify vulnerabilities along a suspicious path detected by the SVR, the test data generation should follow. Given a suspicious path, test data generator is responsible to generate test data to make the execution of the program under test to follow it. We have developed two different approaches for test data generation:

- Test data generation using Moped reachability checker and program slicing. The detail of this approach is discussed in Chapter 4;
- Hybrid approach for test data generation. This approach utilizes the power of static and dynamic analysis for test data generation. The details of this approach are discussed in Chapter 5.

3.6.3 Dynamic Vulnerability Detector

With the test data generated, Dynamic Vulnerability Detector (DVD) [99] can execute the instrumented program under test and then monitor the execution to collect concrete evidences of the violation of security properties. Once the violation of security properties is

detected, DVD stops the execution, or suppresses the action depending on the user configuration, then generates a test report, which details where and when vulnerabilities occur. For example, for memory leak, it can trace the actual memory address and the size of the leak.

In general, DVD provides the following functionalities:

- Allows a user to specify the security properties with a graphical interface: the security specifications are expressed as edit automata. DVD provides a graphical interface to let the analysts visually draw state diagrams. A user can define the states of the finite state machine, the transition between the states as well as the transition guard, which determines whether or not an event will trigger the transition. After that, the system transforms the visual representation of the finite state machine into a language used by the State Machine Compiler (SMC) [27]. The SMC then generates the C++ code, which is a shared library with C interfaces, to represent the state machines;

- Instruments code to the program under test: In order to monitor the execution of the program under test, code instrumentation is needed. DVD injects code fragments into the program under test so as to synchronize the state machine and the program under test. By checking if the state machine reaches an error state, DVD detects whether a violation has occurred;

- Generates test report: once a violation is detected, DVD gathers the dynamic information during the execution of the program under test, and generates the report. The report contains information such as the type of the detected flaws and locations of

these vulnerabilities.

3.7 Summary

In this chapter, we proposed a hybrid vulnerability detection approach that covers the security properties specification, potential vulnerability detection, test data generation, and dynamic verification. In addition, we introduced our extended GCC for code instrumentation. By combining the static and dynamic analysis, we take advantages of both of them so as to increase the efficiency and accuracy of the vulnerability detection. In our approach, static analysis is used to find all the possible vulnerable sites of a program, and test data generation and dynamic analysis are utilized to verify the possible vulnerabilities so as to reduce the false positives.

Chapter 4

Test Data Generation Using Moped

This chapter introduces the test data generation approach using Moped reachability checker and program slicing. Model-checkers have recently been suggested for automated software test-case generation. However, the use of a model-checker comes at the price of potential performance problems that are linked to state explosion [50]. If the model used for test-case generation is complex, then model-checker based approaches can be very slow, or even not applicable at all. To mitigate the problem, we propose a new approach that combines the Moped reachability checker with program slicing for test data generation.

4.1 Moped Reachability Checker

Moped reachability checker is a model checker for pushdown systems [69]. The checker simulates program execution based on the Remopla [81] language, which is used to model

a program, for all possible arguments within a finite range and generate coverage information for these executions. Given a target in a Remopla representation, the checker attempts to verify whether, or not, that target is reachable. If the target is reachable, then the checker would generate a trace, including test data values, which would lead the execution to the target. The formal language used by Moped reachability checker is a state transition system, where each possible value for each variable is represented by a state. As a result, it faces the state space explosion problem. Consequently, there is a large performance overhead for these generations to be made. Generally, the performance of the model checker is affected by the range of the variables and the complexity of the program, such as number of variables, number of lines of code and number of function calls, etc.

The checker, as designed, is not suitable for our purpose of security test data generation since it is concerned with producing data to reach a specific target without any regard to a specify path. For our system, we require test data to be generated for a specified path; that is reaching the target through any path is insufficient. Consequently, we need to simplify the model, which is a Remopla representation, used by the checker so that it contains only the code fragments along the suspicious path. Additionally, we need to enhance the performance of the checking process. To achieve these goals, we resort to program slicing to optimize the Remopla representation. Subsection 4.2 details the program slicing component of our model.

4.2 Program Slicing

Program slicing was introduced by Mark Weiser in 1981 [93], as a method of abstracting programs and reducing them to a minimal form that is still capable of producing an original subset of behavior. A slicer is an executable program that performs identical actions to a specified subset of the original program. Depending on how minimal a slice is desired, a slicing criterion is set, which specifies a window for observing the program behavior.

In our approach, the slicer works on the GIMPLE representation of the program under test instead of the original source code. Our decision to work on GIMPLE representation was driven by the following factors: first, GIMPLE contains control flow and data flow information, which is necessary for slicing. Second, some optimization has been done by GCC before generating GIMPLE representation, such as partial redundancy elimination, dead code and unreachable code elimination.

Slices are produced by computing consecutive sets of transitively relevant statements based on the control flow and data flow information. Given the control flow information, we identify and remove any basic block that is not related to the suspicious path. This results into a subset of the original GIMPLE representation, which contains only blocks along the suspicious path. Following that, using the data flow information, we identify and remove all unrelated variables as well as reductant code. Figure 10 illustrates a GIMPLE representation containing five basic blocks. However, the suspicious path contains only basic blocks 0, 1, 4. First we remove those basic blocks that are not related to the path, which results in the removal of basic block 2 and 3 from the GIMPLE representation.

Second, unused variables like $x2$, $x2.1$ in the basic blocks are eliminated. Figure 11 shows the resulting GIMPLE representation.

4.3 Architecture

The architecture of test data generation utilizing Moped reachability checker and program slicing is shown in Figure 12. This architecture contains six components that are:

- XML-Dump [45]: an extended GCC that dumps GIMPLE representation in XML format;
- XML-Parser: an XML parser written in C++, with which object-oriented tree representations of GIMPLE are constructed from the XML files generated by XML-Dump;
- Static Analyzer: it scans the GIMPLE tree and gathers control flow and data flow information along a suspicious path. It then passes the information to the program slicer;
- Program Slicer: it simplifies the GIMPLE representation into a minimal, with a slicing criterion window that includes the suspicious path and the set of relevant variables to that path;
- Program Abstractor: it converts the sliced GIMPLE representation of source code into Remopla model;

```
main ()
{
 int x2;
 int x1;
 char filename[12];
 int D.2652;
 int x2.1;
 int x1.0;

 # BLOCK 0
 # PRED: ENTRY (fallthru)
 filename = "thefile.txt";
 scanf (&"%d"[0], &x1);
 scanf (&"%d"[0], &x2);
 x1.0 = x1;
 if (x1.0 == 100) goto <L0>; else goto <L1>;
 # SUCC: 1 (true) 2 (false)

 # BLOCK 1
 # PRED: 0 (true)
<L0>:;
 access (&filename, 2);
 open (&filename);
 goto <bb 4> (<L3>);
 # SUCC: 4 (fallthru)

 # BLOCK 2
 # PRED: 0 (false)
<L1>:;
 x2.1 = x2;
 if (x2.1 == 200) goto <L2>; else goto <L3>;
 # SUCC: 3 (true) 4 (false)

 # BLOCK 3
 # PRED: 2 (true)
<L2>:;
 open (&filename);
 # SUCC: 4 (fallthru)

 # BLOCK 4
 # PRED: 1 (fallthru) 2 (false) 3 (fallthru)
<L3>:;
 D.2652 = 0;
 return D.2652;
 # SUCC: EXIT
}
```

Figure 10: GIMPLE Representation before Slicing

```
main ()
{
 int x1;
 char filename[12];
 int D.2652;
 int x1.0;

 # BLOCK 0
 # PRED: ENTRY (fallthru)
 filename = "thefile.txt";
 scanf (&"%"[0], &x1);
 x1.0 = x1;
 if (x1.0 == 100) goto <L0>; else goto <L1>;
 # SUCC: 1 (true) 2 (false)

 # BLOCK 1
 # PRED: 0 (true)
<L0>:;
 access (&filename, 2);
 open (&filename);
 goto <bb 4> (<L3>);
 # SUCC: 4 (fallthru)

 # BLOCK 4
 # PRED: 1 (fallthru) 2 (false) 3 (fallthru)
<L3>:;
 D.2652 = 0;
 return D.2652;
 # SUCC: EXIT
}
```

Figure 11: GIMPLE Representation after Slicing

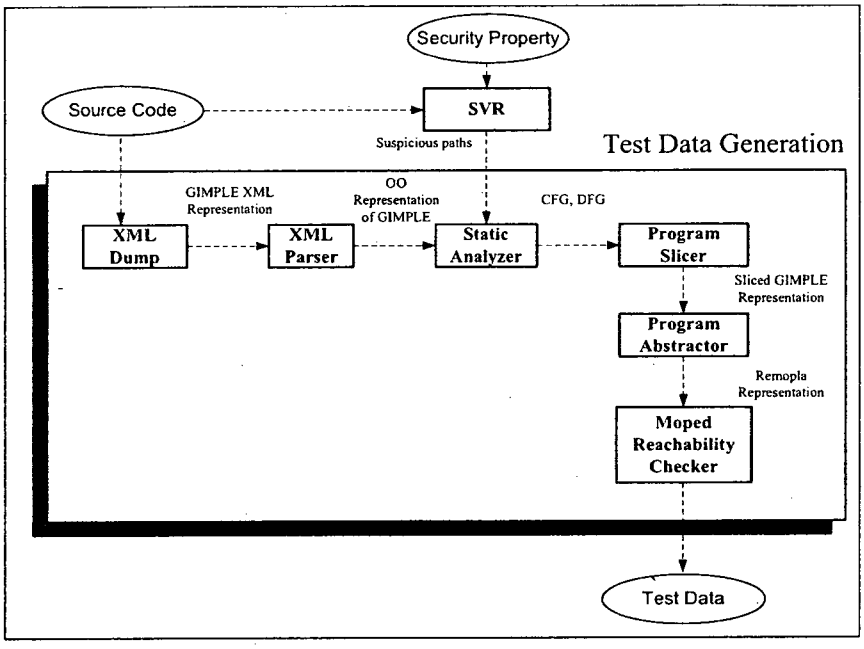


Figure 12: System Architecture of Moped Test Data Generator

- Moped Reachability Checker: given a Remopla model, this component is responsible for the generation of test data.

The test data generation process can be summarized as follows: given source code and a suspicious path, the XML-Dump and XML-Parser work together to generate object-oriented GIMPLE tree representation of the program under test. The static analyzer then performs static analysis to gather control flow and data flow information that are related to a suspicious path. This information is then given to the program slicer that produces a sliced GIMPLE representation. This representation is then injected into the program abstractor, which transforms it to a Remopla model. Finally, the generated Remopla model is used as an input to the Moped reachability checker for the generation of test data.

4.4 Approach Evaluation

With program slicing technique, the Remopla model used by the Moped reachability checker is simplified. Consequently, the performance of the checking process can be improved. To evaluate the approach, we generated test data for a sample program listed in the Appendix A before slicing and after slicing. As shown in Figure 13, program slicing results in large performance enhancement, especially when the state space of variables is increased.

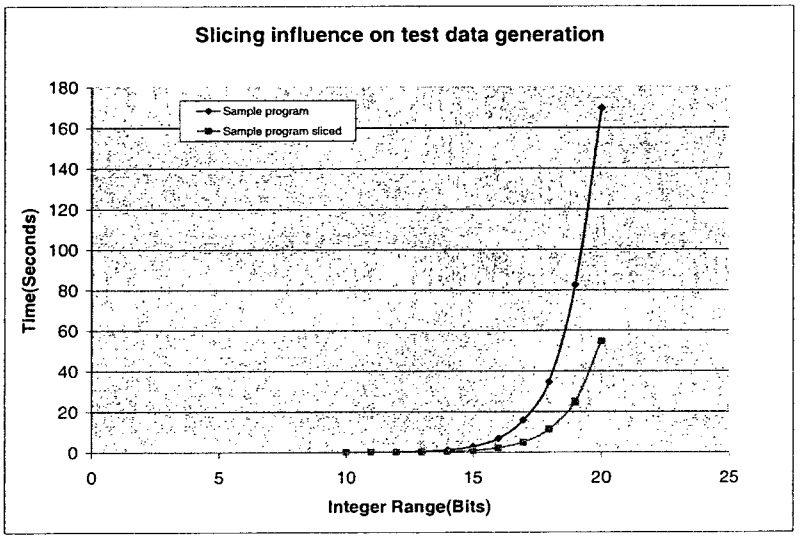


Figure 13: Slicing Influence on Test Data Generation

This approach is limited by the Moped reachability checker itself, which suffers from state space explosion problem. Therefore, the performance of this approach is affected by the complexity of the program under test. Furthermore, the degree of simplicity of sliced programs depends on the coupling of programs along the suspicious path with the rest parts

of the program under test.

4.5 Summary

In this chapter, we presented the architecture for test data generation, which takes advantages of Moped reachability checker and program slicing. Due to the limitation of this approach, it mainly targets safety-critical software of relatively smaller sizes, such as embedded software. In order to expand the flexibility and scalability of our vulnerability detection framework, we propose a hybrid approach for test data generation in chapter 5.

Chapter 5

Hybrid Approach to Test Data Generation

This chapter introduces a hybrid approach for test data generation, which uses the information gathered form static and dynamic analysis to guide the test data generation process. Before proceeding further with the discussion of the hybrid approach to test data generation, we need to define the following terms:

Definition 1. *Problem node: a basic block, where the execution of a program under test follows an undesired branch.*

Definition 2. *Conditional basic block: a basic block that ends with a conditional statement.*

Definition 3. *Execution path: a sequence of basic blocks:* $path = \langle bb_1, bb_2, ..., bb_n \rangle$.

Definition 4. *Suspicious path: an execution path that might contain violations of security properties.*

Definition 5. *Desired branch: a branch of a conditional statement, which takes the execution of the program under test to follow the suspicious path.*

Definition 6. *Undesired branch: any branch of a conditional statement, which leads the execution away from the suspicious path.*

Definition 7. *Controlling variables: variables appearing in a conditional statement.*

Definition 8. *Constraint function: a function constructed from the conditional statement. A conditional statement has the following format: lhs op rhs, where the lhs represents the operand on the left-hand side, the rhs represents the operand on the right-hand side, and the op represents the comparison operator, such as* $<, \leq, >, \geq, \neq, =$. *The format of the constraint function depends on the op and the branch that the execution needs to follow.*

Definition 9. *Constraint value: the concrete value of a constraint function calculated dynamically.*

5.1 General Description of the Approach

For our purpose of test data generation, our interest is to generate test data for a suspicious path rather than a program point. Accordingly, our test data generation approach should be a path-oriented one. During the survey of test data generation techniques, we found that Korel's approach [70] is a promising one. Korel's approach is a dynamic one, which solves some problems of symbolic execution in handling loops and array. Our approach is an extension of it. Our model starts by converting all the constraints along a suspicious

path into constraint functions. We define $c(x)$ as a constraint function, which is derived from the conditional statements. It is formulated as follows: $c(x)\{<, \leq, =, \neq\}0$, where x is the set of input variables along the path. Table 7 illustrates how to construct a constraint function from a conditional statement. The idea behind this is to make constraint function be relative to zero.

Conditional Statement	Desired Branch	Constraint Function $c(x)$	Satisfied Constraint Value
lhs $>$ rhs	True branch	rhs $-$ lhs	< 0
lhs $>$ rhs	False branch	lhs $-$ rhs	≤ 0
lhs $>=$ rhs	True branch	rhs $-$ lhs	≤ 0
lhs $>=$ rhs	False branch	lhs $-$ rhs	< 0
lhs $<$ rhs	True branch	lhs $-$ rhs	< 0
lhs $<$ rhs	False branch	rhs $-$ lhs	≤ 0
lhs $<=$ rhs	True branch	lhs $-$ rhs	≤ 0
lhs $<=$ rhs	False branch	rhs $-$ lhs	< 0
lhs $=$ rhs	True branch	lhs $-$ rhs	$= 0$
lhs $=$ rhs	False branch	lhs $-$ rhs	$\neq 0$
lhs $\neq$ rhs	True branch	abs(lhs $-$ rhs)	< 0 or > 0
lhs $\neq$ rhs	False branch	abs(lhs $-$ rhs)	$= 0$

Table 7: Constraint Function Table [70]

Once a problem node is encountered during the test data generation, the subgoal of test data generation becomes to minimize the constraint function of the problem node so as to overcome the problem node. To this end, the test data generation can now be viewed as a minimization problem. Since dynamic analysis has runtime overhead, the minimization process must be as fast as possible.

By observing the dynamic behavior of the constraint functions, reasonable assumptions can then be made as what the relation is between $c(x)$ and x. The constraint value is calculated in reference to zero. For example, consider a program node with the following

condition: $if(x > 12)$, and where the true branch is part of the suspicious path. Now, suppose at dynamic time, the initial generated data for x is 0. The constraint value is calculated as $(12 - x) = 12$. Our target is to make this constraint value less than 0, so that the desired branch is executed. The following attempts would then try to make this constraint value less than 0. In addition, through the observation, we can estimate whether the constraint function $c(x)$ is linear or non-linear in order to decide what technique should be used to generate test data.

Test Data Generation based on Linear Programming

For linear programming, we use the one-dimensional search procedure [53], which consists of two steps, exploratory search and pattern search, to guide the iteration of test data generation. In the first step, an exploratory move is used to determinate the direction, in which to proceed. In this step, one input variable is selected to be increased and decreased by a small offset, while keeping other variables constant. For each change, we execute the program and calculate the constraint value of the problem node if the execution fails to follow the suspicious path. By comparing the values of the input variables and the constraint values, a direction is decided. Afterwards, the pattern search is started. During this search process, a large move to the input variable is made and the program is executed. Once the violation is detected, and the constraint value is moving towards the satisfied value, a larger move is made in the same direction.

In our implementation, the observation process starts by the execution of three attempts with an initial guess x_k, $x_k - \Delta$ and $x_k + \Delta$, where Δ is a small offset. The outcome is used

to determine which direction we should proceed. If the direction is detected, we adjust the input and start the execution of the program under test again. The general algorithm for test data generation for linear functions is illustrated in Algorithm 1.

Algorithm 1 Test Data Generation Algorithm based on Linear Programming

Input: instrumented program under test, suspicious path
Output: test data

$x \Leftarrow initial\_guess$
while a problem node encountered && resource is not exhausted **do**
  //Execute the instrumented program with test data x, and gather the constraint values
  $c(x) = execute(x)$
  if *suspicious_path* is reached **then**
    $return\ x$;
  end if
  /*Adjust the test data with a small offset,execute the instrumented program and gather the constraint values */
  $c(x + \Delta) = execute(x + \Delta)$
  if *suspicious_path* is reached **then**
    $return\ x$;
  end if
  /*Adjust the test data with a small offset in another direction,execute the instrumented program and gather the constraint values */
  $c(x - \Delta) = execute(x - \Delta)$
  if *suspicious_path* is reached **then**
    $return\ x$;
  end if
  //Exploratory move is used to determine the direction, in which to proceed
  $direction = calculate\_direction((x + \Delta), (x - \Delta), c(x + \Delta), c(x + \Delta))$;
  //A large move to the input variable is made
  $move = calculate\_move(previous\_move, direction)$;
  $x = x + move$;
end while

Test Data Generation based on Non-linear Programming

If the relationship between the input data x, and the $c(x)$ is non-linear, then Newton-Raphson method, which is effective in finding a minimum or maximum of a real-valued

function [33], is utilized. Our interest is to find a value with which the constraint function will reach beyond the threshold of zero. By attempting quick convergence to a minima, we do achieve our goal of crossing the threshold quickly.

Given a constraint $c(x)$ and its derivatives $c^{'}(x)$ and $c^{''}(x)$, we begin with a first guess x_0. According to Newton-Raphson method: the approximation x_1 is calculated as follows:

$$x_1 = x_0 - \frac{c^{'}(x)}{c^{''}(x)}$$

As the details of the constraint function is unknown, instead of computing $c^{'}$ and $c^{''}$ exactly, we can use finite difference approximations:

$$c^{'}(x) \approx \frac{c(x+\Delta) - c(x-\Delta)}{2\Delta}$$

;

$$c^{''}(x) \approx \frac{c(x+\Delta) - 2c(x) + c(x-\Delta)}{\Delta^2}$$

,where the Δ is a small offset. According to Newton-Raphson method, we adjust the test data as follow:

$$x_{k+1} = x_k - \frac{\Delta}{2} \frac{c(x_k+\Delta) - c(x_k-\Delta)}{c(x_k+\Delta) - 2c(x_k) + c(x_k-\Delta)}$$

The general algorithm for test data generation for non-linear functions is illustrated in

Algorithm 2.

Algorithm 2 Test Data Generation Algorithm based on Non-linear Programming

Input: instrumented program under test, suspicious path
Output: test data

$x \Leftarrow initial\_guess$
while problem node encountered && resource is not exhausted **do**
//Execute the instrumented program with test data x, and gather the constraint values
$c(x) = execute(x)$
if *suspicious_path* is reached **then**
$return\ x$;
end if
/*Adjust the test data with a small offset, execute the instrumented program
and gather the constraint values*/
$c(x + \Delta) = execute(x + \Delta)$
if *suspicious_path* is reached **then**
$return\ x$;
end if
/*Adjust the test data with a small offset in revers direction,
execute the instrumented program and gather the constraint values*/
$c(x - \Delta) = execute(x - \Delta)$
if *suspicious_path* is reached **then**
$return\ x$;
end if
//The test data is adjusted based on Newton-Raphson method
$x = x - \frac{\Delta}{2} \frac{c(x+\Delta)-c(x-\Delta)}{c(x+\Delta)-2c(x)+c(x-\Delta)}$;
end while

5.2 Static Analysis Phase

Our approach starts the static analysis phase first to analyze the GIMPLE representation of the source code of the program under test. GIMPLE contains the control flow and data flow information needed for static analysis. We use an XML patch [45] of GCC, which dumps the GIMPLE of the program under test into XML files. To facilitate the analysis of the

GIMPLE, we design and implement a set of C++ classes, which has a similar structure to GIMPLE. Figure 14 shows a fragment of the class diagram. After dumping the XML files, we parse them with an XML parser written in C++, and then create the object-oriented GIMPLE representation, based on which the static analysis can be conducted.

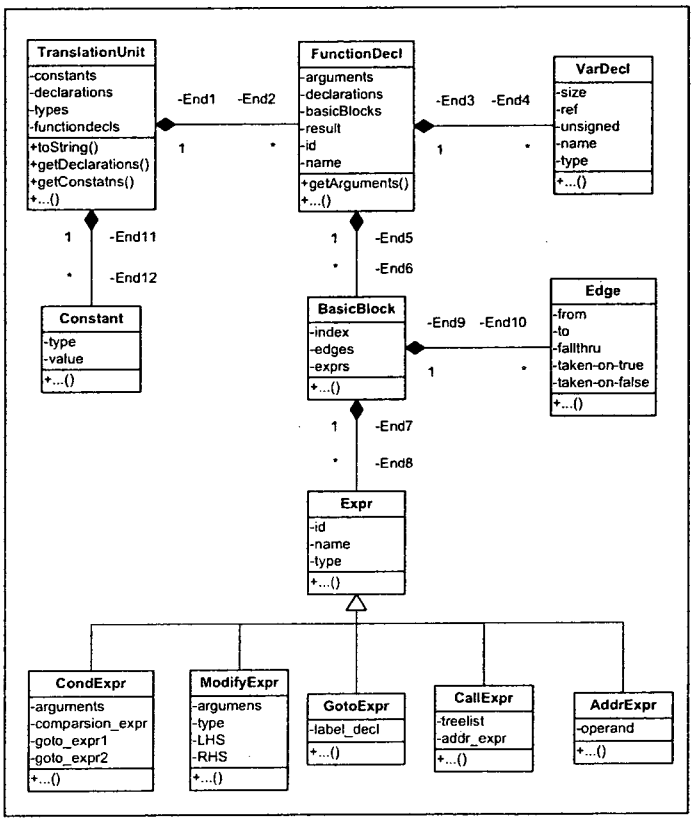


Figure 14: Partial Class Diagram of GIMPLE

5.2.1 Control Flow Analysis

In this analysis, we classify the branches of each conditional statement along a suspicious path into two categories: desired and undesired. Any branch that leads the execution away

from the suspicious path, will be classified as undesired, otherwise as desired. Once the execution of the program under test hits an undesired branch, it will be halted immediately.

5.2.2 Data Flow Analysis

The following data flow information along a suspicious path are also gathered:

- Input variables along the suspicious path;
- Controlling variables along the suspicious path;
- Dependencies between the input variables and the controlling variables.

The purpose of the data flow analysis is to analyze the dependency between controlling variables and input variables. The input variables are the variables whose values are needed to be generated so that to make the execution of the program under test follow the suspicious path. The controlling variables are variables appearing in a conditional statement along the path.

After the input variables and controlling variables information is gathered, the dependency analysis is conducted. The purpose of this dependency analysis is that: once a problem node is encountered during test data generation, we just need to adjust the input variables that affect the constraint function inside that problem node, leaving those unrelated input variables unchanged, so as to speed up the test data generation process. For example, in Figure 15, our interest is to generate test data to reach the `malloc` statement. There are ten input variables along the suspicious path, and the constraint function `1098-ctrl_var` of the problem node is controlled by the the variable `ctrl_var`, which is

dependent on the input variable `x10`. As a result, only test data for `x10` will be adjusted. There is no need to generate test data for the rest input variables. Therefore, the number of iterations is reduced. As a result, the performance of the test data generation is enhanced.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
        int x1,x2,x3,x4,x5,x6,x7,x8,x9,x10;
        char *buffer;
        scanf("%d%d%d%d%d%d%d%d%d%d",&x1,&x2,&x3,&x4,
                &x5,&x6,&x7,&x8,&x9,&x10);
        int ctrl_var = x10 + 123;
        if(ctrl_var > 1098){
                buffer = malloc(1024);
        }else if(x1 > 1024 && x2 < -897 && x3 == 12345){
                ...
        }else if(x4 == 789 && x5 > 5324 && x6 < -97){
        }else if(x7 == 145 & x8 == 79 && x9 == 459){
                ...
        }else{
                ...
        }
        return 0;
}
```

Figure 15: Data Flow Analysis Sample

A controlling variable is dependent on an input variable if and only if:

1. The controlling variable itself is the input variable;

2. The constraint function is the use of the input variable.

The static analysis aims to improve the efficiency of the dynamic analysis that is costly in terms of runtime overhead. The enhancements are as follows: first, only test data of the related input variables will be generated, which results in less number of iterations. Second, once the execution of instrumented program under test hits a undesired branch, the

execution is halted.

5.3 Dynamic Analysis Phase

In the dynamic analysis phase, dynamic information, such as constraint value, problem node, and execution history, is collected.

5.3.1 Monitoring for Test Data Generation

In order to conduct dynamic analysis, the runtime behavior of programs under test needs to be monitored. The following monitoring functionalities are needed for the purpose of test data generation:

- Program execution monitoring: our approach is an iterative approach. The test data generation attempts are adjusted based on the constraint values, which are obtained by monitoring the runtime value of the constraint functions along the suspicious path. In addition, once the program execution follows an undesired branch, the monitoring code should stop the execution immediately and identify the problem node;

- Test data communicating: the test data generation logic is implemented in test data generation server, which will be detailed in Section 5.4.2. The test data generation process involves the communication between the program under test and the server. The test data is generated by the server, and sent to the client, which is the program under test. At the same time, the client collects dynamic information, such as constraint value, execution history, during its execution, and send them to the server. To

enable the communication between the client and the sever, the test data communicating code fragments need to be injected into the program under test;

- User input abstracting: since our system strives to achieve the maximum automation possible, we need to abstract program statements that would require user interaction (i.e., `scanf()`). The input abstractor replaces the input statements, so that calls for data entries are abstracted by other calls that would perform the same functionality without any user interaction. Currently, our implementation provides an abstraction to only a limited set of all possible input methodologies, but extensions can be added.

5.3.2 APIs for Monitoring

The calls of the following APIs are instrumented into the program under test for the monitoring purpose:

- `_initialize()`: initializes the monitoring functionalities, such as allocating memory and creating log files;
- `_finalize()`: finalizes the monitoring, such as releasing the memory and writing log files;
- `_do_execution_monitoring(int bb_index, const char* funcName)`: halts the program once the execution hits a undersized path;
- `_calculate_constraint(int lhs, int rhs, int oper, int bb_index, const char* funcName)`: calculates the constraint value before the execution of the conditional statement;

- `_abstract_input(const char* format, ... )`: extracts the environmental interaction, such as user input statement, and automate the test data input process;
- `_send_msg(char* msg)`: sends messages to the server;
- `_receive_msg(char* msg)`: receives messages from the server.

5.4 Execution Management

For dynamic analysis, we need to restart the execution of the program under test, once an attempt fails. At the same time, we need to store the information collected from the static and dynamic analysis and compute test data based on the collected information. In order to protect the testing environment, it is better to separate the test data computation and the execution of the program under test. To achieve this goal, we introduce the execution manager, which is a TCP/IP socket client-server framework. It has two major components: instrumented programs under test that serves as a client, and a test data generation server as illustrated in Figure 16. This design also provides the flexibility to generate test data for more than one programs at the same time.

5.4.1 Instrumented Program Under Test

An instrumented program under test serves as a client to the test data generation server. Once a user interaction is needed, it will send a request to the test data generation server, and use the response as the input. In addition, the client is responsible for collecting dynamic

information and sending them to the server. On the other hand, the instrumented program under test is executed under monitoring. During the execution, once it hits an undesired branch, it will be halted immediately.

5.4.2 Test Data Generation Server

We need a component that can store the information collected from static and dynamic analysis, perform data analysis, and generate test data. A test data generation server is introduced for that purpose. The communication between the server and a client is based on TCP/IP socket. Figure 16 illustrates the communication steps involved during test data generation process between the sever and the client.

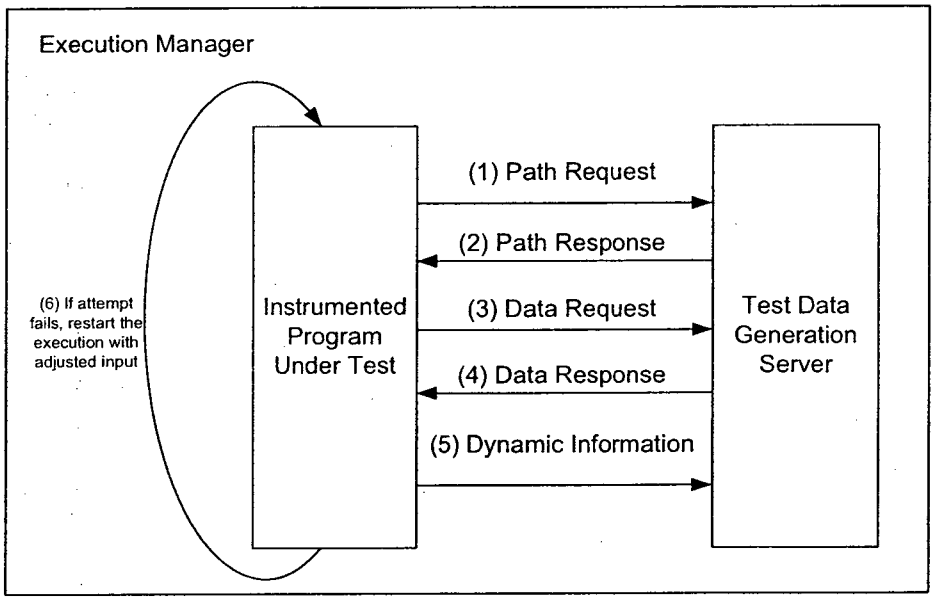


Figure 16: Execution Manager

The communication steps involved are as follows:

- Path request: When the client executes, it needs to know the suspicious path, which

it should follow. The path request is formatted as follows: *program_name:path_request*;

- Path response: the server will send the path information to the client once it receives the request. The path has a format that is as follows: *path_response:function_name:*bb_1, *function_name:*bb_2, ..., *function_name:*bb_n;

- Data request: during the execution, when a user input is needed, user input abstracting function sends data request to the server automatically. The format of the data request is as follows: *data_request:function_name:*bb_n*:variable_name:line_num*;

- Data response: the sever generates test data based on the collected information. It then sends it to the client. The response is constructed as follows: *data_response:function_name:*bb_n*:variable_name:line_num:value* ;

- Dynamic information feedback: during the execution, the client gathers dynamic information and sends it to the server. Such dynamic information includes: constraint values and execution history. To send a constraint value, the following message is constructed and sent:*constraint_value:function_name:*bb_n*:value*. The execution history message has a format as follows: *path_response:function_name:*bb_1, *function_name:*bb_2, ..., *function_name:*bb_n.

The test data generation is based on the static and dynamic information collected from the program under test. The test data generation server uses hash tables to store such information, which are illustrated as follows:

- Branch Direction Table
- Dependency Table
- Constraint Value Table
- Test Data Generation History Table

Branch Direction Table

Table 8 contains branch directions of every node that has a conditional statement. It is constructed during static analysis. The left column of the table represents basic blocks and their branches. The right column shows the classification of the branches. The purpose of using this table is to identif unnecessary runtime cost. Once the execution of the client hits an undesired branch, the execution can be halted immediately so as to save dynamic analysis time.

Basic Block Number	Branch Direction
bb_1:true_branch	desired/undesired
bb_1:false_branch	desired/undersired
...	...
bb_n:true_branch	desried/undesried
bb_n:false_branch	desried/undesried

Table 8: Branch Direction Table

Dependency Table

Table 9 illustrates the dependency between input variables and conditional basic blocks along the suspicious path. It shows which input variables are affecting which basic blocks. This table is constructed during the static analysis phase. The first column of the table

contains basic blocks, the second column is a set of input variables, which might affect the controlling variables in the conditional statement of the basic blocks. This table is used to optimize the test data generation process. When a problem node is encountered, only the value of the input variables that have dependency on the problem node will be adjusted for next attempts, which results in less number of iterations. In addition, this table helps detect some infeasible paths. If a problem is encountered during the test data generation process for a suspicious path and the input variable set for the problem node is null, which means that this basic block is not controlling by any input variables, we say the path is infeasible.

Basic Block Number	Input Variable
bb_1	$\{x_1, x_2, ..., x_n\}$
bb_2	$\{y_1, y_2, ..., y_n\}$
...	...
bb_n	$\{z_1, z_2, ..., z_n\}$

Table 9: Controlling Variable Table

Constraint Value Table

The constraint value of a problem node is very important for us to check how far away the test data is from satisfaction. To make the execution follow the suspicious path, all constrain value along the path must be a satisfied constraint value as shown in Table 7. If a constraint value of a node is not satisfied, it means that node is a problem node. This information is gathered during the dynamic time.

Basic Block Number	Constraint Value
bb_1	val_1
bb_2	val_2
...	...
bb_n	val_n

Table 10: Constraint Value Table

Test Data Generation History Table

The test data generation server keeps track of the test generation history for each input variable. The next test data generated is based on the last data attempted, and the constraint value of the problem node. Table 11 contains the test attempted for each input variable along the suspicious path. The first column of this table is the input variable, and the test data that has been attempted is shown in the second column. Once a successful test data generation is achieved, the last test data attempt is the satisfied one.

Input Variable	Value
x_1	$val_1, val_2, ..., val_n$
x_2	$val_1, val_2, ..., val_n$
...	...
x_n	$val_1, val_2, ..., val_n$

Table 11: Test Data Generation History Table

5.5 System Design

The system design of the hybrid test data generator is illustrated in Figure 17. This generator contains six components:

- XML-Dump: it is an extended GCC, with which the GIMPLE representation of a program under test will be written to files in XML format;

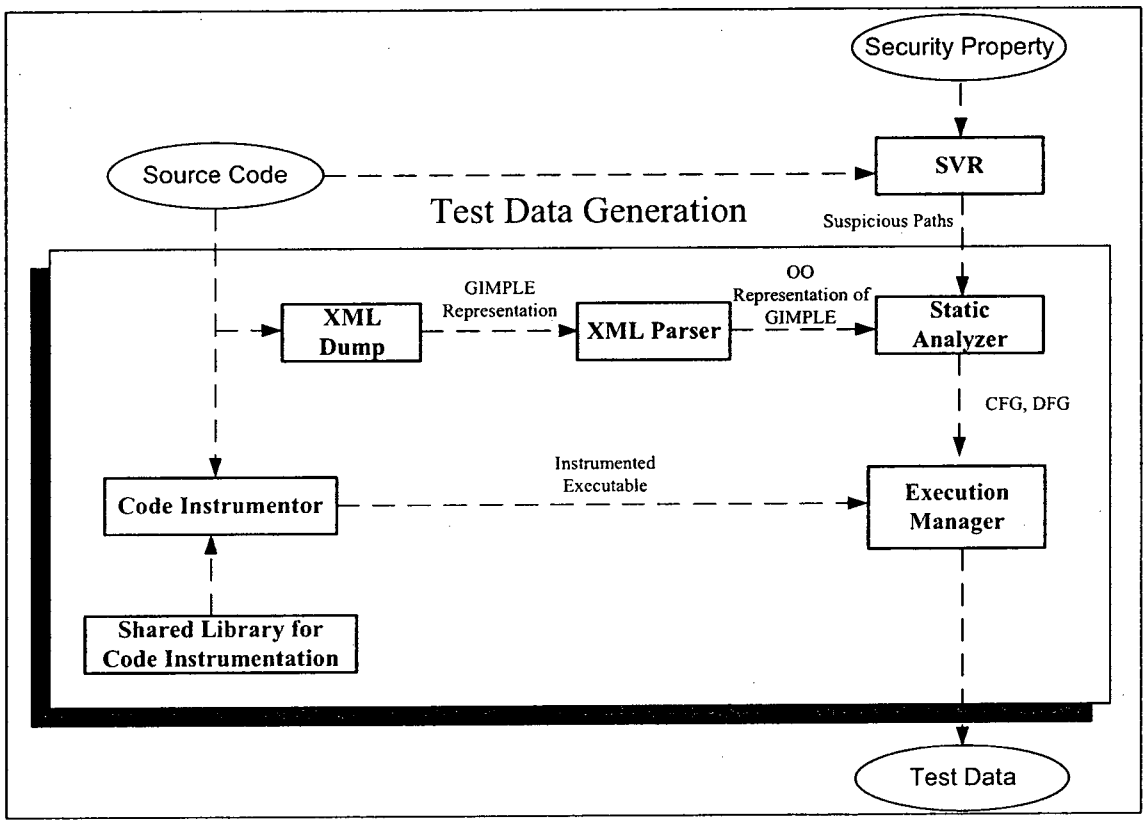


Figure 17: System Design of Hybrid Test Data Generator

- XML-Parser: it is an XML parser written in C++, with which object-oriented representations for the GIMPLE are constructed from the XML files injected by the XML-Dump;
- Static Analyzer: it analyzes the GIMPLE representation and gathers control flow and data flow information along the suspicious path;
- Code Instrumentor: it is an extended GCC that instruments monitoring code into the program under test;
- Shared Library for Instrumentation: this contains the implementations of the APIs for monitoring, calls of which are instrumented into the program under test;
- Execution Manager: it is responsible to coordinate the test data generation server and its client (the instrumented program under test).

Briefly, this approach is an iterative approach. The test data generation process can be summarized as follows: starts with the XML-Dump, source code of the program under test is translated into GIMPLE representation in XML format, with which the XML-Parser constructs object-oriented representation of the GIMPLE. Given the output from XML-Parser, the static analyzer performs static analysis to collect control flow and data flow information along a suspicious path. All the static information will be stored in the test data generation server. On the other hand, the code instrumentor, an extended GCC 4.2, injects calls of the APIs for monitoring into the program under test during the compilation. After that, the execution manager executes the instrumented program under test, which sends dynamic information to the test data generation server and waits for test data from it.

With the collected dynamic and static information, test data generation server generates test data, and sends it back to the client. Once the attempt fails, the execution manager restarts the instrumented program under test and the server will adjust the test data according to the collected static and dynamic information.

5.6 Summary

In this chapter, we explained the test data generation process that utilizes the synergy between the static and dynamic analysis. In this approach, we convert the test data generation problem into a minimization one. We are inspired by Korel's approach. However, our approach goes beyond Korel's one. The distinctions are as follows:

- This approach is designed for security testing purpose, while Korel's approach is designed for general purpose of test data generation;
- In our approach, Newton-Raphson method is used to improve the efficiency of test data generation;
- Code instrumentation is done automatically. We extend the GCC 4.2 to automate the code instrumentation for test data generation;
- Our approach is a hybrid approach, which utilizes the static analysis to guide the dynamic analysis;
- We target C language in our research, while Korel is interested in Pascal. Our approach can be extended to any language that is supported by GCC.

Chapter 6

Integrated System and Experiments

This chapter introduces the interfaces and functionalities of the integrated system for test data generation. The experimental results with the integrated system are also presented.

6.1 Integrated System

This section introduces the GUI of our integrated system for test data generation.

6.1.1 System Overview

Figure 18 illustrates the overview of the GUI of the integrated system. The GUI can be divided into three areas. First, the upper part is a menu bar. By clicking the menu on it, a user can choose what subsystem to be used. There are three main subsystems listed in the menu bar, which are SVR, Automatic Test Data Generator (ATDG), and DVD. Second, a project list is shown in the left panel. For any software under test, we create a project

for it so that the source code of the software can be imported into the system. Third, the right panel is a console to display system messages. Any feedback information from the execution of any task will be displayed in this panel. We mainly discuss the GUI related with ATDG, which is our implementation for test data generation.

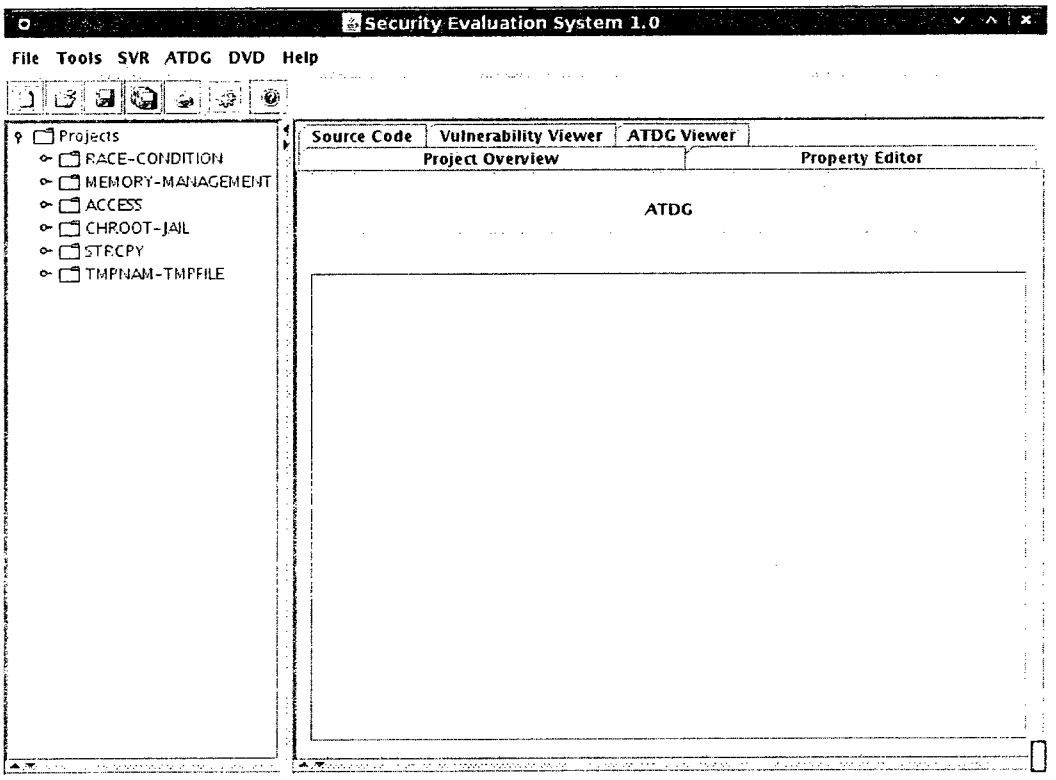


Figure 18: System Overview

6.1.2 System Configuration

Before using the system for test data generation, some system-specific settings for test data generation must be done. Figure 19 illustrates the configurations to be set at the initial use. These settings include:

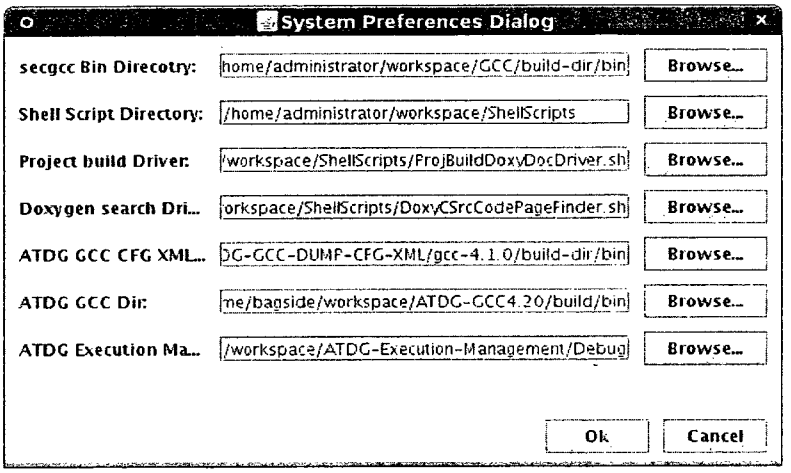


Figure 19: System Configuration

- Location of the shell scripts, which helps invoking the code written in C or C++ commands from Java;
- Location of the XML dumper, which is used for dumping the GIMPLE representation of the program under test into XML files;
- Location of the extended GCC 4.2 for code instrumentation;
- Location of the execution manager, which launches the test data generation server and the instrumented executable of a program under test.

6.1.3 Interfaces

Figure 20 shows the main interfaces of ATDG. It contains the following items:

- Create Project: imports the source files of the program under test into the system.
- Delete Project: removes a project from the project list;

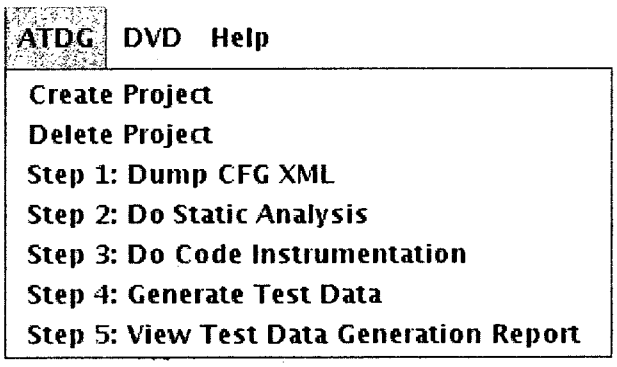


Figure 20: System Menu

- Dump CFG XML: generates GIMPLE representation in XML format;
- Do Static Analysis: conducts static analysis over the program under test;
- Do Code Instrumentation: instruments the calls of the APIs for monitoring into the program under test;
- Generate Test Data : executes the instrumented program under test to generate the test data;
- View Test Data Generation Report: displays the generated test data report.

6.1.4 Test Data Generation

Once the code instrumentation is done, as illustrated in Figure 21, a user can execute the instrumented executable by selecting the executable file and defining the suspicious path. Given an instrumented executable, and a suspicious path, the test data generation process is initialized.

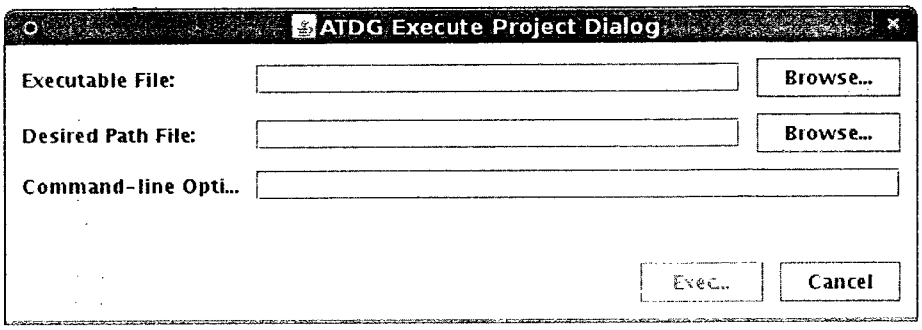


Figure 21: Test Data Generation Dialog

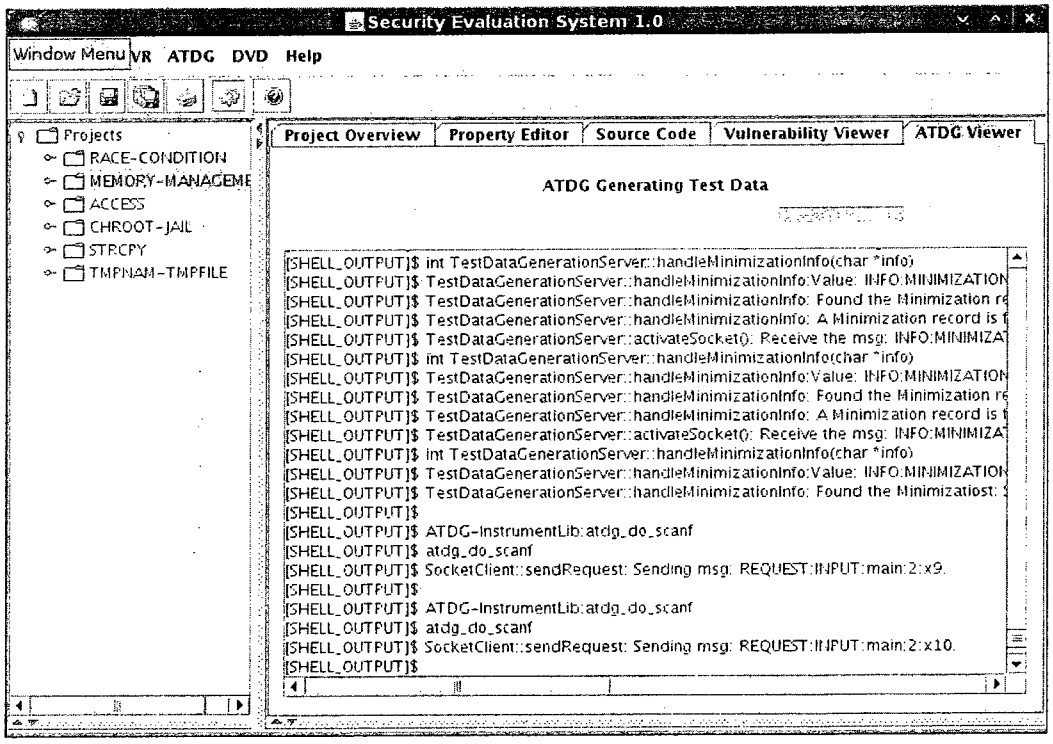


Figure 22: Test Data Generation

Figure 22 illustrates the dynamic analysis process for generating the test data for the suspicious path.

6.1.5 Test Data Generation Report

When the test generation is successfully done, a test data generation report will be generated as a HTML file. Figure 23 shows a sample of test data generation report. The report contains path information, test data generation history, and statist information, such as the time used.

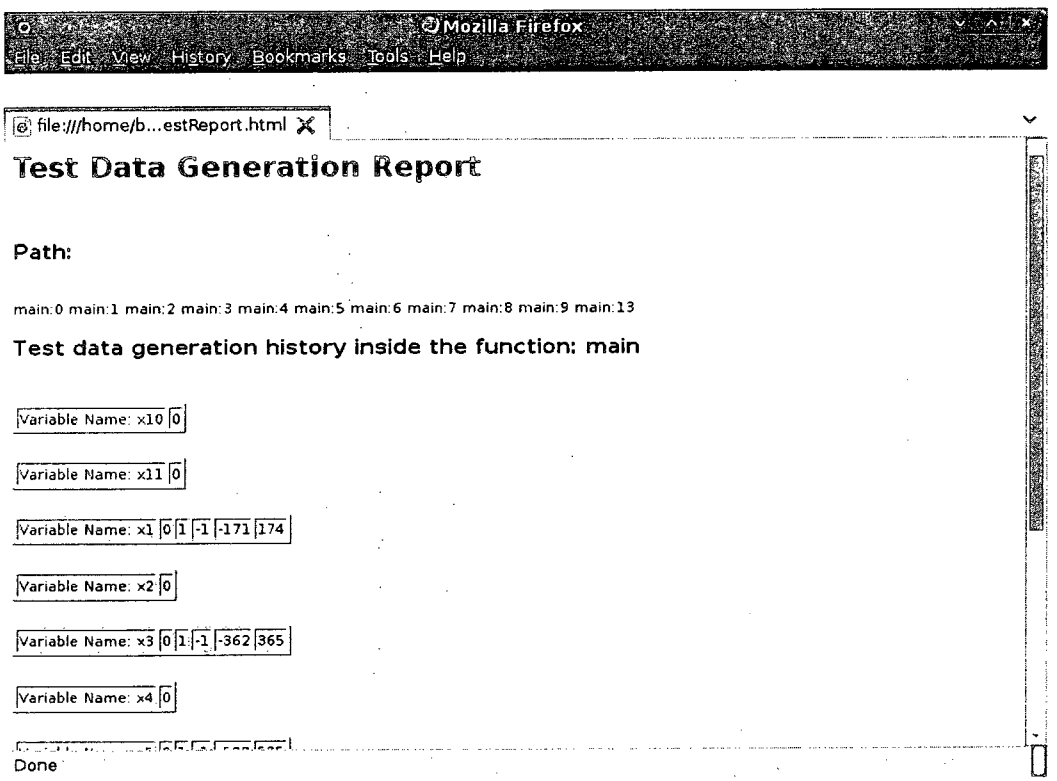


Figure 23: Test Data Generation Report

6.2 Experiments

The experiments emphasize the aim of applying our framework to test suites of programs.

6.2.1 Environment

All experiments were run on a core-2 PC with a clock frequency of 2.0 GHZ, 2 MB of level II cache, and 2G RAM under the following environments:

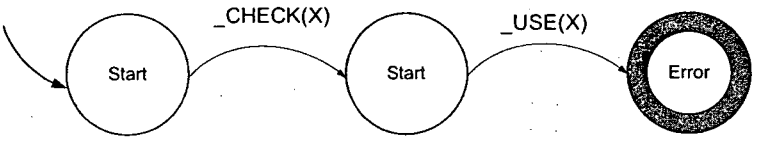
- Operating system: Ubuntu Linux release 6.10
- C/C++: GCC 4.2
- Java runtime environment: Java(TM) SE Runtime Environment (build 1.6.0 02b05)
- Moped: Moped version 2

6.2.2 Validated Security Properties

During the experiments, we define a set of security properties to be validated.

1. RACE-CONDITION: Time-of-check-to-time-of-use (TOCTOU) vulnerabilities are due to the fact that there is a period of time between the checking of a condition and the use of the results of that check. This period of time allows either an attacker to intentionally or another interleaved process or thread to unintentionally change the state of the targeted resource and yield unexpected and undesired results [65]. TOCTOU vulnerabilities are usually a kind of race condition. To prevent TOCTOU race conditions that might be exploited by an attacker to substitute the file between

the check (e.g., `stat` or `access` call) and the use (`open` call), a program should not pass the same file name to two system calls on any path. Figure 24 shows the automaton that represents the security property RACE-CONDITION.



_CHECK(X) = access, stat, statfs, statvfs, lstat, readlink, tempnam, tmpnam, tmpnam_r

_USE(X) = acct, au_to_path, basename, catopen, chdir, chmod, chown, chrrot, copylist, creat, db_initalize, dbm_open, dbminit, dirname, dlopen, execl, execle, execlp, execv, execve, execvp, fattach, fdetach, fopen, freopen, ftok, ftw, getattr, krb_recvauth, krb_set_tkt_string, kvm_open, lchown, link, mkdir, mkdirp, mknod, mount, nftw, nis_getservlist, nis_mkdir, nis_ping, nis_rmdi, nlist, open, opendir, patchconf, pathfind, realpath, remove, rename, rmdir, rmdirp, scandir, symlink, system, t_open, truncate, umount, unlink, utime, utimes, utmpname

Figure 24: RACE-CONDITION Automaton

2. CHROOT-JAIL: the `chroot()` function establishes a virtual root directory for the owning process. The main purpose of `chroot()` is to confine a user process to a portion of the file system so as to prevent unauthorized access to system resources. Programs like `ftp` and `httpd` commonly make use of this function. The `chroot()` function requires root (super-user) access to call. If the programmer continues to run as root after the `chroot()` call, he or she opens up a potential vulnerability window for an attacker to use elevated privilege. Another problem of `chroot()` is that it changes the root directory but not the current directory. Therefore, program can escape from changed root if it forgets calling `chdir("/")`. Figure 25 shows

the automaton that represents the security property CHROOT-JAIL.

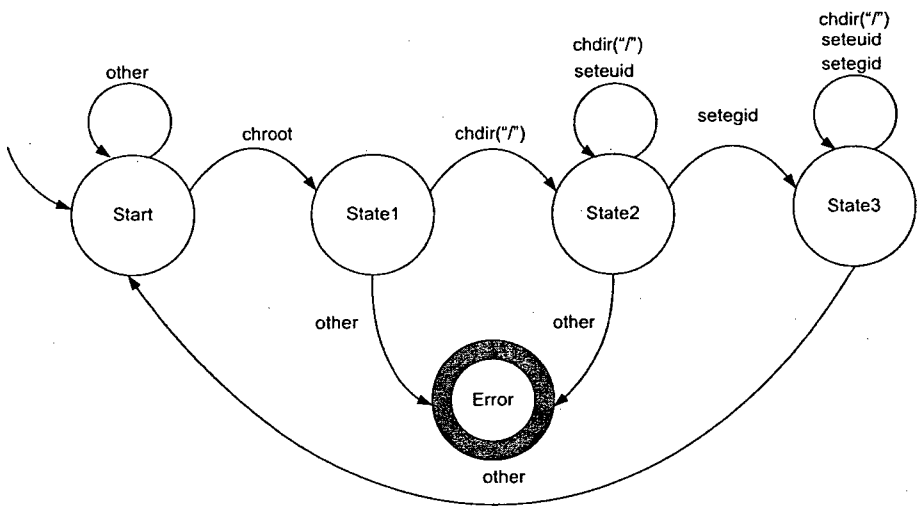


Figure 25: CHROOT-JAIL Automaton

3. MEMORY-MANAGEMENT: in this security property, we define four states: `start`, `allocated`, `deallocated`, and `error`. When a memory pointer is created, the state of the pointer will be updated as `start`. Once the pointer is assigned to a memory location, the state will be transited to `allocated` from `start`. If the pointer is released, the state will be transited from `allocated` to `deallocated`. Any transition leads to the error state represents a violation of the security property. Figure 26 shows the automaton that represents the security property MEMORY-MANAGEMENT. This automaton is capable of catching the following errors:

   - Memory leak: this error occurs when a program fails to release memory when

no longer needed. In the automaton, assigning a memory location to an uninitialized pointer will transit the state of the pointer from `start` to state `allocated`. The exit of the program without freeing the pointer will then transit the state into the `error` state, which indicates a memory leak error is caught;

- Double free: this error will be triggered when the release operation is applied more than once on the same memory address. In the automaton, once the pointer is in the `deallocated` state, any release operation on that pointer will trigger the double free error;
- Using uninitialized memory: it means the use of the pointer that has not been allocated a memory address yet. In the automaton, if the current state is `start`, any read or write operation will trigger this error. In addition, any read or write operation on the pointer that is in the `deallocated` state will also trigger this error.

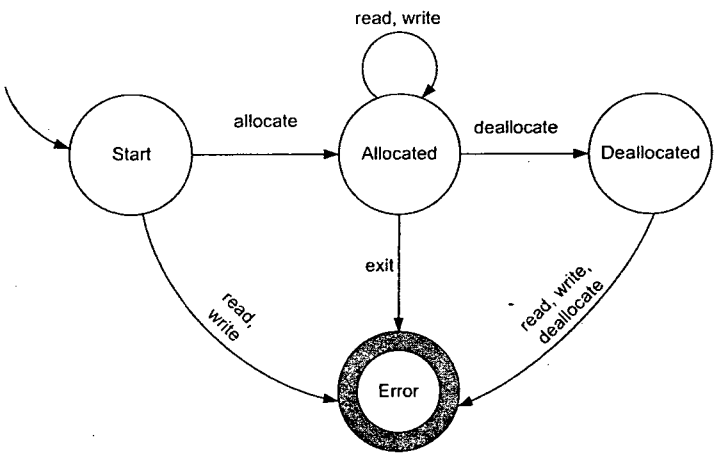


Figure 26: MEMORY-MANAGEMENT Automaton

4. STRCPY: it is drawn from the coding rule STRCPY of CERT [65]. The string copy library functions are vulnerable to buffer overflow attacks. The `strcpy(dest, src)` function is the classic one that is vulnerable to buffer overflow attacks. To use this function properly, the destination buffer must be big enough to hold the source string plus the null terminating character. Otherwise, it gives the attacker an opportunity to send input larger than the buffer size, which results in overflowing the buffer. This can be exploited by an attacker to implement a denial of service (DoS) or buffer overflow attack [65]. SVR is not able to detect buffer overflow vulnerabilities like this, since string length comparison and string content checking can only be conducted dynamically. However, SVR can report suspicious paths where the vulnerable function `strcpy` is used. A buffer-overflow security can be classified as non-reachability property. To reduce it to a reachability one, additional constraints should be added. In this case, to prove the exist of the violation, additional constraint {`strlen(src) >= strlen(dest)`} or {`dest[n-1]` $\neq$ '\0'} is added as additional constraint to the constraint set along the suspicious path. Then, test data generation is used to verify whether or not a violation of buffer-overflow exists. Once the test data, which satisfies the constraints along the path as well as {`strlen(src) >= strlen(dest)`} or {`dest[n-1]` $\neq$ '\0'}, is generated, we prove that the violation is positive. Figure 27 shows the automaton that represents the security property STRCPY.

5. TEMPNAM-TMPFILE:

   very often, software applications use temporary files for information sharing, and

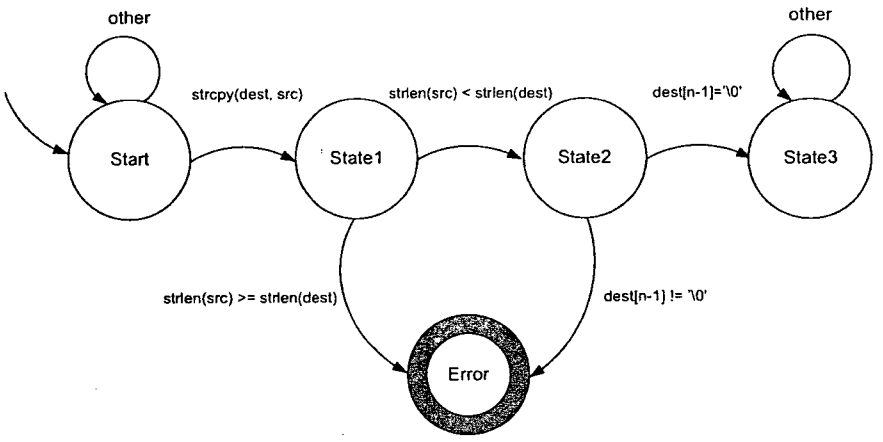


Figure 27: STRCPY Automaton

temporary data storing. However, many applications use the temporary files in a loose way: terminating execution without cleaning these temporary files, which gives attackers a chance to hijack private and sensitive data. Therefore, the temporarily files must be used with caution. The standard C library provides a set of functions for temporary files creation. However, some of these functions are vulnerable to various forms of attacks and must be used with precaution. This security property is drawn based on the coding rule TEMPNAM-TMPFILE from CERT [65]. This security property focus on two aspects, temporary file creation and permissions management. Temporary file names created by the `tmpnam` family of functions such as `tmpnam()`, `tempnam()`, `tmpfile()`, and `mktemp()` can be easily guessed by an attacker. Incorrect temporary file creation can lead to TOCTOU and accessibility vulnerabilities [65]. As a result, these functions should not be used. On the other hand, as a temporary file is usually created in a shared folder, the appropriate

permissions should be set to these files so to ensure the protection against attackers. As such, a call to `umask(077)` must be done before a call to `mkstemp` to make sure that only the owner can access these temporary file. With considering these two factors, we draw an automaton to represent this security property as is illustrated in Figure 28.

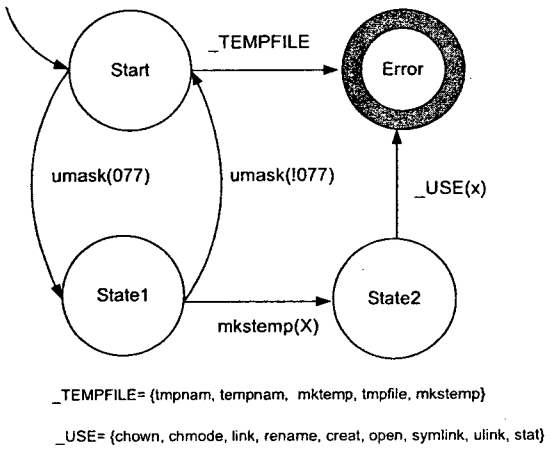


Figure 28: TEMPNAM-TMPFILE Automaton

6.2.3 Programs Under Test

We conduct the experiments on a suite of C programs, which contain violations of the above-mentioned security properties. This suit contains the following five programs under test:

1. Program 1: this program has a RACE-CONDITION vulnerability, which is triggered when the program calls a check function: `access` followed by a use function:

`open`;

2. Program 2: this program contains a violation of CHROOT-JAIL security property. The program has a vulnerable path, where the `chroot` is used in a incorrect way;

3. Program 3: this program has a MEMORY-MANAGEMENT vulnerability. Memory leak occurs when the execution terminates without releasing allocated memory;

4. Program 4: this program is vulnerable to buffer-overflow attack. It violates the security property STRCPY;

5. Program 5: this program manages temporarily files using vulnerable functions. It has a violation of TEMPNAM-TMPFILE security property.

The details of the programs under test are listed in Appendix B.

6.2.4 Results

Given the security properties, SVR detected the following set of suspicious paths from the five programs under test:

- Program 1: tested against the security property RACE-CONDITION, and a suspicious path is detected as follow: $(main : 0)$ $(main : 1)$ $(main : 2)$ $(main : 3)$ $(main : 5)$ $(main : 13)$ $(main : 14)$ $(main : 15)$ $(main : 16)$ $(main : 17)$;

- Program 2: tested against the security property CHROOT-JAIL, and a suspicious path is detected as follow: $(main : 0)$ $(main : 1)$ $(main : 2)$ $(main : 3)$ $(main : 4)$ $(main : 5)$ $(main : 6)$ $(main : 17)$;

Program name	Security property	Time (millisecond)	Iterations
Program 1	RACE-CONDITION	329325	151
Program 2	CHROOT-JAIL	5234254	2057
Program 3	MEMORY-MANAGEMENT	95520	175
Program 4	STRCPY	27249942	9763
Program 5	TEMPNAM-TMPFILE	17239761	6560

Table 12: Experimental Results using Random Testing

- Program 3: tested against the security property MEMORY-MANAGEMENT, and a suspicious path is detected as follow: $(main : 0)$ $(main : 1)$ $(main : 2)$ $(main : 3)$ $(main : 4)$ $(main : 5)$ $(main : 6)$ $(main : 7)$ $(main : 8)$ $(main : 9)$ $(main : 13)$;

- Program 4: tested against the security STRCPY, and a suspicious path is detected as follow: $(main : 0)$ $(main : 1)$ $(main : 2)$ $(main : 3)$ $(main : 4)$ $(main : 5)$ $(main : 6)$ $(main : 8)$ $(main : 9)$ $(main : 10)$ $(main : 11)$ $(main : 12)$ $(main : 23)$;

- Program 5: tested against the security TEMPNAM-TMPFILE, and a suspicious path is detected as follow:$(main : 0)$ $(main : 1)$ $(main : 2)$ $(main : 3)$ $(main : 4)$ $(main : 5)$ $(main : 6)$ $(main : 8)$ $(main : 9)$ $(main : 10)$ $(main : 11)$ $(main : 12)$ $(main : 13)$ $(main : 14)$ $(main : 15)$ $(main : 16)$ $(main : 17)$ $(main : 18)$ $(main : 19)$ $(main : 20)$ $(main : 21)$ $(main : 22)$ $(main : 25)$.

In order to verify the existence of the vulnerabilities, test data generation is performed. For test data generation, four different approaches are utilized: random testing, Moped reachability checker without program slicing, Moped reachability checker with program slicing, and the hybrid approach. The experimental results are shown in Tables 12, 13, 14. Two measurements were used to evaluate the performance of the different approaches: the

Program name	Security property	Time without slicing (millisecond)	Time with slicing (millisecond)	Enhancement (%)
Program 1	RACE-CONDITION	5580	250	96%
Program 2	CHROOT-JAIL	56740	28000	51%
Program 3	MEMORY-MANAGEMENT	12190	20	99.8%
Program 4	STRCPY	370	190	49%
Program 5	TEMPNAM-TMPFILE	80	70	12.5%

Table 13: Experimental Results using Moped Reachability Checker

total time used for the whole test data generation process, and the number of iterations, which means how many times the program under test is executed (not applicable to Moped reachability checker).

As shown in Table 12, in a total of five programs under test, random testing approach is able to generate test data. However, the time consumed for each case varies dramatically. The result shows that the performance of random approach for test data generation is unstable. The time for test data generation is unpredictable. Whenever there is an "equal to" condition along the suspicious path, the performance of random testing decreases a lot. Theoretically, the random approach is able to generate test data for any feasible suspicious path since it is a matter of time. However, in reality, time matters.

Table 13 shows the experiment results of test data generation based on Moped reachability checker without and with program slicing. For small size programs, these two approaches can generate test data very quickly. The experimental results also show that the program slicing improves the efficiency of test data generation. However, this approach (Moped reachability checker with program slicing) has drawbacks. First, its performance is limited by state space. As the state space goes up, the performance drops down. Second,

Program name	Security property	Time (millisecond)	Iterations
Program 1	RACE-CONDITION	1413	13
Program 2	CHROOT-JAIL	1493	13
Program 3	MEMORY-MANAGEMENT	5757	12
Program 4	STRCPY	140800	37
Program 5	TEMPNAM-TMPFILE	502532	121

Table 14: Experimental Results using the Hybrid Approach

as the complexity (such as the number of the variables, number of sub components) of the program under test increases, the performance decreases dramatically. Third, the enhancement of program slicing is affected by the program under test itself. Once the code along a suspicious path has high-level of coupling with other code in the program, the improvement through program slicing is not obvious. As a result, this approach is not suitable for large size software, which usually have complex function calls and high level of coupling. Therefore, we suggest it to be used for embedded safety-critical software that usually has a relatively small size.

The experimental result Table 14 shows that the hybrid approach is a stable one. It can generate test data in a reasonable time. The number of iterations and the time used depend on factors like number of constraints of a suspicious path, the execution time of the program under test, the number of code instrumented, and the relationship (linear or non-linear) between input variables and control variables. Usually, linear programming takes more time than non-linear one. Different from Moped reachability checker, the performance of this approach is not affected by the state space. In addition, the time consumed increases in linear rather than exponential way as the number of constrains grows.

The test data generation experimental results show that random testing has the lowest efficiency in generating test data. Test data generation based on Moped reachability checker suffers from the state space explosion problem, which limits it to be used for small size safety-critical software. Among these three different approaches, hybrid is the most promising approach for test data generation. All the test data generation results prove that all the vulnerabilities detected by SVR are not false-positives. The overall experience shows that the system works as envisioned. Experiments clearly indicate that the system is capable of performing vulnerability detection in a systematic way: from specification to detection and verification.

Chapter 7

Conclusion

This chapter concludes our thesis. First, we summarize our contributions, then we describe the future as an extension to our work.

The goal of our research is to automate the process of vulnerability detection so that it can be conducted in a systematic way: from security property specification to potential vulnerability spotting and verification. The aim of the automation is to improve the efficiency and accuracy of the security vulnerability detection.

In this thesis, we described a framework which utilizes the synergy between the static and dynamic analysis for vulnerability detection. The basic idea of this framework is to use static analysis to point out the potential vulnerabilities and verify them with dynamic analysis. To link the static analysis and dynamic analysis, test data generation is conducted to generate test cases, which will make the execution of the program under test to reach a vulnerable site following a suspicious path. With test cases generated, dynamic analysis can be performed to verify the vulnerabilities detected. Inside this framework, security

properties are specified by security automata, which can describe a wide range of security properties. This framework spots potential vulnerabilities by using Moped model-checker with automatically generated models, which are synchronized with the security automata. The test data generation is done through two different approaches: Moped with program slicing approach and hybrid approach. Moped with program slicing approach is suitable for small size software with security concern, such as embedded software. Hybrid approach is more suitable for complex softwares. In addition, we discussed how to extend GCC 4.2 for code instrumentation. Our framework has some appealing properties that we describe hereafter:

1. Modular design: the major components of our framework, such as the potential vulnerability detection, test case generation, and dynamic verification, are designed as modules. This design leaves space for future upgrade. For example, if there is a better technique to spot potential vulnerability, our potential vulnerability detection component can be replaced without affecting the other parts of the framework.

2. Automation: inside this framework, the process of potential vulnerability detection is automated as well as the test data generation and dynamic verification. In addition, the code instrumentation is done during compilation without user interaction.

3. Flexibility: our framework is based on GIMPLE representation, which is language-independent. Hence, our framework can be extended to support multiple programming languages that are supported by GCC.

4. False positive reduction: our framework reduces the number of false positives by

generating test data to dynamically assess vulnerabilities.

We also introduced the idea of reducing security vulnerability detection to reachability. For most cases, reachability is enough to prove the violation of a security property. However, for a few cases, such as buffer-overflow, only providing reachability to vulnerable site is not sufficient. Therefore, we proposed the concept of injecting additional constraint to the program under test to make sure that the test data generated will trigger the violation of the security property under concern. By applying this concept, we reduce the security vulnerability detection into reachability.

So far, our system has built up a preliminary infrastructure for security vulnerability detection with number of false positives reduced. We used our tool to detect security vulnerability in a set of C programs which contain different vulnerable code. The vulnerabilities detected are verified as positives. Our experiments validated our design.

In order to achieve the goal of full automation, the following issues need to be addressed:

1. Support more data types: other types of inputs should be considered, such as string and float;

2. Support more languages: the current implementation focuses on C code. We need to extend our approach to support more programming languages;

3. Improve the data flow analysis: for precise data flow analysis, pointer alias analysis should also be included;

4. Improve the efficiency of test data generation by incorporating quasi-Newton methods into the hybrid approach.

Ultimately, this security testing framework is meant to be coupled with a systematic security hardening component of free and open source software. By doing so, we will have an end-to-end system that starts from source code, detect vulnerabilities and fix them in a systematic way.

Bibliography

[1] BOON. http://www.cs.berkeley.edu/~daw/boon/. (Date of Access: June 11, 2009).

[2] Build Security in. https://buildsecurityin.us-cert.gov/daisy/bsi/home.html/. (Date of Access: June 11, 2009).

[3] CodeWizard. http://www.parasoft.com. (Date of Access: June 4, 2009).

[4] Computer Emergency Response Team. http://www.cert.org/stats/. (Date of Access: July 18, 2008).

[5] Coverity Prevent. http://www.coverity.com/. (Date of Access: June 4, 2009).

[6] Cppcheck. http://cppcheck.wiki.sourceforge.net/. (Date of Access: June 11, 2009).

[7] Daikon. http://groups.csail.mit.edu/pag/daikon/. (Date of Access: June 4, 2009).

[8] Dmalloc. http://dmalloc.com/. (Date of Access: July 27, 2007).

[9] EXE. http://www.stanford.edu/~engler/. (Date of Access: June 11, 2009).

[10] GCC Core 4.2.0. http://gcc-ca.internet.bs/releases/gcc-4.2.0/. (Date of Access: August 20, 2007).

[11] GIMPLE. http://www.gimpel.com/. (Date of Access: July 27, 2007).

[12] The GNU Compiler Collection. http://gcc.gnu.org/. (Date of Access: August 24, 2007).

[13] History of viruses. http://www.computer-sleuth.com/history_sof_sviruses.htm. (Date of Access: July 18, 2008).

[14] Holodeck. http://www.securityinnovation.com/holodeck/index.shtml. (Date of Access: June 4, 2009).

[15] Icat statistics. http://icat.nist.gov/icat.cfm?function=statistics, 2007.

[16] Insure++. http://www.parasoft.com. (Date of Access: June 4, 2009).

[17] ITS4. http://www.cigital.com/its4/. (Date of Access: June 04, 2009).

[18] Klocwork. http://www.klocwork.com/. (Date of Access: June 04, 2009).

[19] Mc Checker. http://suif.stanford.edu/research/analysis.html. (Date of Access: July 27, 2007).

[20] Netcraft. http://news.netcraft.com. (Date of Access: June 04, 2009).

[21] OWASP. http://www.owasp.org/index.php/Category:Vulnerability. (Date of Access: June 3, 2009).

[22] Peach Fuzzer. http://peachfuzzer.com/. (Date of Access: June 4, 2009).

[23] PScan. http://seclab.cs.ucdavis.edu/projects/testing/tools/pscan.html. (Date of Access: June 14, 2009).

[24] Rational PurifyPlus. http://www-01.ibm.com/software/awdtools/purifyplus/. (Date of Access: June 04, 2009).

[25] SPIKE. http://www.immunitysec.com/resources-freesoftware.shtml. (Date of Access: June 3, 2009).

[26] Splint. http://www.splint.org/. (Date of Access: June 4, 2009).

[27] State machine compiler. http://smc.sourceforge.net/. (Date of Access: May 13, 2009).

[28] Valgrind. http://valgrind.org/. (Date of Access: June 04, 2009).

[29] IEEE standard glossary of software engineering terminology, 10 Dec. 1990.

[30] The Open Source Definition. http://www.opensource.org/, July 2006.

[31] ANSI/IEEE. Standard Glossary of Software Engineering Terminology. Technical report, 1991.

[32] Cyrille Artho, Viktor Schuppan, Armin Biere, Pascal Eugster, Marcel Baur, and Boris Zweimüller. JNuke: Efficient dynamic analysis for java. In Rajeev Alur and Doron

Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 462–465. Springer, 2004.

[33] Roberto Battiti. First- and second-order methods for learning: between steepest descent and newton's method. *Neural Computation*, 4(2):141–166, 1992.

[34] BBC. Monster attack steals user data. `http://news.bbc.co.uk/2/hi/technology/6956349.stm`, August 2007.

[35] Thoms Bell. The concept of dynamic analysis. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 216–234, London, UK, 1999. Springer-Verlag.

[36] Janis Bicevskis, Juris Borzovs, Uldis Straujums, Andris Zarins, and Edward F. Miller. Smotl a system to construct samples for data processing program debugging. *IEEE Transactions on Software Engineering*, 5(1):60–66, 1979.

[37] David L Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM systems journal*, 22(3):229–245, 1983.

[38] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, 1975.

[39] Mandira Chakraborty and Uday K. Chakraborty. An analysis of linear ranking and binary tournament selection in genetic algorithms. *Information, Communications and*

Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on, 1:407–411 vol.1, Sep 1997.

[40] Hao Chen and David Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, Washington, DC, november 2002.

[41] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[42] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[43] P. David Coward. Symbolic execution and testing. *Information and Software Technology*, 33(1):53–64, 1991.

[44] Nachum Dershowitz and Zohar Manna. Inference rules for program annotation. *IEEE Transactions on Software Engineering*, 7(2):207–222, 1981.

[45] Dietmar Ebner. GIMPLE to XML patch. `http://www.complang.tuwien.ac.at/cd/ebner/`. (Date of Access: February 2, 2009).

[46] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, October 1999.

[47] Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *In WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[48] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Software Engineering Methodol.*, 5(1):63–86, 1996.

[49] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.

[50] Gordon Fraser and Franz Wotawa. Using LTL rewriting to improve the performance of model-checker based test-case generation. pages 64–74, 2007.

[51] Free Software Foundation, http://gcc.gnu.org/onlinedocs/gccint/. *The GCC Internals*, 1.2 edition, 2007.

[52] Moheb R. Girgis. An experimental evaluation of a symbolic execution system. *Software Engineering Journal*, 7(4):285–290, 1992.

[53] H. Glass and L. Cooper. Sequential search: A method for solving constrained optimization problems. *Journal ACM*, 12(1):71–82, 1965.

[54] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[55] Michael Goulde. Open source becoming mission-critical in north america and europe. `http://www.forrester.com/Research/Document/Excerpt/0,7211,38866,00.html`, September 2006.

[56] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. *SIGSOFT Software Engineering Notes*, 23(6):231–244, 1998.

[57] Rachid Hadjidj, Xiaochun Yang, Syrine Tlili, and Mourad Debbabi. Model-checking for software vulnerabilities detection with multi-language support. In *PST '08: Proceedings of the 2008 Sixth Annual Conference on Privacy, Security and Trust*, page 133Ű142, Washington, DC, USA, 2008. IEEE Computer Society.

[58] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69–82, New York, NY, USA, 2002. ACM.

[59] Jon Heffley and Pascal Meunier. Can source code auditing software identify common vulnerabilities and be used to evaluate software security? In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90277, Washington, DC, USA, 2004. IEEE Computer Society.

[60] Bill Hetzel. *The complete guide to software testing (2nd ed.)*. QED Information Sciences, Inc., Wellesley, MA, USA, 1988.

[61] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[62] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[63] William E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[64] Marieke Huisman and Alejandro Tamalet. A formal connection between security automata and jml annotations. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 340–354, Berlin, Heidelberg, 2009. Springer-Verlag.

[65] Build Security in. Coding Rules. `https://buildsecurityin.us-cert.gov/daisy/bsi-rules/home/g1/848-BSI.html`, April 2007.

[66] Fortify Software Inc. Rough Auditing Tool for Security. `http://www.fortify.com/security-resources/rats.jsp`. (Date of Access: July 27, 2007).

[67] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Comput.*, 44(2):248–260, 1995.

[68] Cem Kaner. Exploratory testing. *Quality Assurance Institute Worldwide Annual Software Testing Conference*, November 2006.

[69] Stefan Kiefer, Stefan Schwoon, and Dejvuth Suwimonteerabuth. Moped - a model-checker for pushdown systems. http://www.fmi.uni-stuttgart.de/szs/tools/moped/, 2008.

[70] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[71] Ounce Labs. Ounce. `http://www.ouncelabs.com/`. (Date of Access: Nov 4, 2009).

[72] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software: Practice and Experience*, 24(2):197–218, 1994.

[73] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. Technical Report TR-720-05, Princeton University, 2005.

[74] Gary McGraw and Bruce Potter. Software security testing. *IEEE Security and Privacy*, 2(5):81–85, 2004.

[75] Christoph C. Michael. Risk-based and functional security testing. `https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/testing/255-BSI.html`, 2005.

[76] Christoph C. Michael, Gary E. McGraw, Michael A. Schatz, and Curtis C. Walton. Genetic algorithms for dynamic test data generation. In *ASE '97: Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, page 307, Washington, DC, USA, 1997. IEEE Computer Society.

[77] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[78] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[79] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[80] Diego Novillo. Tree SSA: A New Optimization Infrastructure for GCC. In *Proceedings the GCC Developers Summits3*, pages 181–193, May 25-27 2003.

[81] University of Stuttgart. Introduction to Remopla. `http://www.fmi.uni-stuttgart.de/szs/tools/moped/`, 2008.

[82] Jiantao Pan. Software testing. `http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/`, 1999.

[83] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[84] Chittoor V. Ramamoorthy, S. B. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, 1976.

[85] Fred B. Schneider. Enforceable security policies. *ACM Transaction of Information System Security*, 2000.

[86] Michael Sutton. *Fuzzing*. Addison-Wesley, Boston, 2007.

[87] Jay-Evan J. Tevis. Automatic detection of software security vulnerabilities in executable program files. 2005. Director-John A. Hamilton, Jr.

[88] Syrine Tlili. *Automatic Detection of Safety and Security Vulnerabilities in Open Source Software*. PhD thesis, Concordia University, Montreal, Canada, 2009.

[89] Nigel Tracey, John Clark, John McDermid, and Keith Mander. A search-based automated test-data generation framework for safety-critical systems. pages 174–213, 2002.

[90] Katrina Tsipenyuk, Brian Chess, and Gary Mcgraw. Seven pernicious kingdoms: a taxonomy of software security errors. *Security & Privacy, IEEE*, 3(6):81–84, 2005.

[91] Jeffrey Voas. Fault injection for the masses. *Computer*, 30(12):129–130, 1997.

[92] Andy J. A. Wang. Security testing in software engineering courses. pages F1C–13–18 Vol. 2, Oct. 2004.

[93] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[94] David A. Wheeler. Why open source software / free software (oss/fs, foss, or floss)? look at the numbers!

[95] James A. Whittaker and Herbert H. Thompson. *How to Break Software Security.* (Addison-Wesley).

[96] Wikipedia. Security testing. `http://en.wikipedia.org/wiki/Security_Testing`. (Date of Access: June 14, 2009).

[97] Wikipedia. Vulnerability. `http://en.wikipedia.org/wiki/Vulnerability_(computing)`. (Date of Access: April 28, 2009).

[98] Xiaochun Yang. Automatic detection of security vulnerabilities in source code. Master's thesis, Concordia University, Montreal, Canada, 2009.

[99] Zhenrong Yang. On Building dynamic vulnerability detection system. Master's thesis, Concordia University, Montreal, Canada, 2007.

Appendix A

Samples for Demonstration

This appendix lists a sample C code, its corresponding GIMPLE representation and control flow graph as well as two samples of Remopla code.

A.1 C Code

Listing A.1: Sample Source Code

```
#include <stdio.h>
int main(){
  int array[123], x1;
  scanf("%d",&x1);
  if(x1 > 0 && x1 < 123){
    int i = 0;
    for(i = 0; i < x1; i++)
      array[i] = i;
  }else{
```

```
    printf("out of boundary\n");
  }
  return 0;
}
```

A.2 GIMPLE Representation

Listing A.2: Sample GIMPLE Representation

```
;; Function main (main)

Merging blocks 7 and 8
main ()
{
  int i;
  int x1;
  int array[123];
  int D.1788;
  int x1.3;
  int i.2;
  int x1.1;
  int x1.0;

  # BLOCK 0
  # PRED: ENTRY (fallthru)
  scanf (&"%d"[0], &x1);
```

```
  x1.0 = x1;
  if (x1.0 <= 0) goto <L5>; else goto <L0>;
  # SUCC: 6 (true) 1 (false)

  # BLOCK 1
  # PRED: 0 (false)
<L0>:;
  x1.1 = x1;
  if (x1.1 > 122) goto <L5>; else goto <L1>;
  # SUCC: 6 (true) 2 (false)

  # BLOCK 2
  # PRED: 1 (false)
<L1>:;
  i = 0;
  i = 0;
  goto <bb 4> (<L3>);
  # SUCC: 4 (fallthru)

  # BLOCK 3
  # PRED: 4 (true)
<L2>:;
  i.2 = i;
  array[i.2] = i;
  i = i + 1;
  # SUCC: 4 (fallthru)
```

```
  # BLOCK 4
  # PRED: 2 (fallthru) 3 (fallthru)
<L3>:;
  x1.3 = x1;
  if (i < x1.3) goto <L2>; else goto <L4>;
  # SUCC: 3 (true) 5 (false)

  # BLOCK 5
  # PRED: 4 (false)
<L4>:;
  goto <bb 7> (<L6>);
  # SUCC: 7 (fallthru)

  # BLOCK 6
  # PRED: 0 (true) 1 (true)
<L5>:;
  __builtin_puts (&"out of boundary"[0]);
  # SUCC: 7 (fallthru)

  # BLOCK 7
  # PRED: 5 (fallthru) 6 (fallthru)
<L6>:;
  D.1788 = 0;
  return D.1788;
  # SUCC: EXIT
```

}

A.3 Control Flow Graph

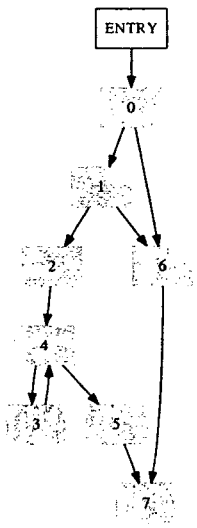


Figure 29: Control Flow Graph of the Sample Code

A.4 Remopla Representation

Listing A.3: Sample Remopla Representation without Slicing

```
define DEFAULT_INT_BITS 20

module void vfork();

module void execve();

init main;

module void main(){
```

```
int x, x1, x2, x3, x4, x5, x6;
if
:: x1 > 1000 ->
  if
  :: x2 > 1000 ->
    vfork();
  :: else -> break;
  fi;
:: else ->
  if
  :: x3 > 1000 -> target1: x=20;
  :: else ->
    if
    :: x4 > 1000 -> target2: x=30;
    :: else ->
      if
      :: x5 > 1000 -> target3: x=40;
      :: else ->
        if
        :: x6 > 1000 -> target4: x=50;
        :: else -> break;
        fi;
      fi;
    fi;
  fi;
fi;
```

```
  target: execve();
}

module void vfork(){}

module void execve(){}
```

Listing A.4: Sample Remopla Representation with Slicing

```
define DEFAULT_INT_BITS 20

module void vfork();
module void execve();

init main;
module void main(){
  int x1, x2;
  if
  :: x1 > 1000 ->
    if
    :: x2 > 1000 ->
      vfork();
    fi;
  fi;
  target: execve();
}
```

```
module void vfork(){}


module void execve(){}
```

Appendix B

Programs Under Test

This appendix lists the programs under test and their CFG

B.1 Program 1

Listing B.1: Program Under Test 1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(){
  int x1,x2,x3,x4,x5,x6,x7,x8,x9,x10;
  char filename[]="thefile.txt";
  scanf("%d%d%d%d%d%d%d%d%d%d",&x1,&x2,&x3,&x4,&x5,&x6,&x7,&x8,&x9);
  x1 = x1 + 104;
  x2 = x2 - 587;
  x3 = 11*x3;
  x4 = x4/12;
  x7 = (x7 + 104)*11/12 - 587;
  x10 = 1023;
  if(x1 > 100 && x2 < 200 && x3 >= 300 ){
    if (0 != access(filename,02)) exit(0);
  }else if(x7 > 700 && x8 > 800 &&x9 > 900){
    //Examples of Corrected Code
    char filename[]="thefile.txt";
    if (0 != seteuid(1200)) { /* handle error */ }
    if (0 != setegid(1200)) { /* handle error */ }
    if (0 != setgroups(0, 0)) { /* handle error */ }
```

```
    FILE *theFile = fopen(filename, "w+");
  }else if(x10 == 1024){
    //infeasible path
    if (0 == access(filename,02)){
      FILE *theFile = fopen(filename, "w+");
    }
  }
  if(x4 <= 400 && x5 != 500 && x6 == 600){
    FILE *theFile = fopen(filename, "w+");
  }
  return 0;
}
```

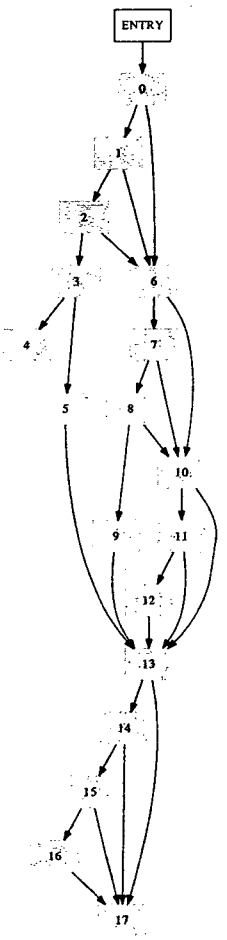


Figure 30: CFG of Program 1

B.2 Program 2

Listing B.2: Program Under Test 2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
//CHROOT-JAIL security property violation
int main(){
  int x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12;
  scanf("%d%d%d%d%d%d%d%d%d%d%d%d",
  &x1,&x2,&x3,&x4,&x5,&x6,&x7,&x8,&x9,&x10,&x11,&x12);
  x2 = x2*x2 - 1000*x2 + 100;
  x7 = x7*x7 - 2000*x7 + 200;
  x12 = 4321;
  if(x1 > 123 && x2 < -50735 && x3 >= 5773 &&
    x4 <= 7834 && x5 == 5527 && x6 != 2778){
    //vulnerable code
    char path[] = "/usr/sandbox";
    chroot(path);
    chdir("..");
    chroot(".");
    execl("bin/sh", "/bin/sh", NULL);
  }else if(x7 == 2711 && x8 == 8439 && x9 < -2874
    && x10 == 483 && x11 == 999){
    //correct way of calling chroot.
    gid_t egid =  getegid();
    uid_t euid = geteuid();
    char path[] = "/usr/sandbox";
    if (chroot(path)) exit(1);
    chdir("/");
    setegid(egid);
    seteuid(euid);
  }else if(x12 != 4321){
    char path[] = "/usr/sandbox";
    chroot(path);
    chdir("..");
    chroot(".");
    execl("bin/sh", "/bin/sh", NULL);
  }
  return 0;
}
```

B.3 Program 3

Listing B.3: Program Under Test 3

```
#include <stdio.h>
#include <stdlib.h>
/* This is the demostration of the violation of
```

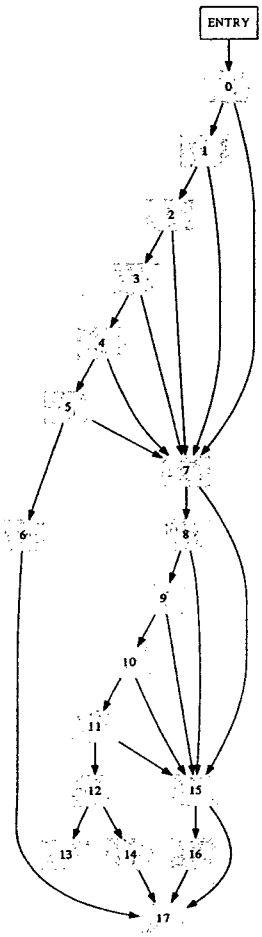


Figure 31: CFG of Program 2

```
the MEMEORY-MANAGMENT security property */
int main(){
  int x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11;
  char *buffer;
  scanf("%d%d%d%d%d%d%d%d%d%d%d",
    &x1,&x2,&x3,&x4,&x5,&x6,&x7,&x8,
    &x9,&x10,&x11);
  x10 = 123456;
  int i;
  if(x1 > 173 && x2 < -257 && x3 >= 365
    && x4 <=469 && x5 ==535 && x6 != 626
    && x7 > 789 && x8 > 888 && x9 > 975 ){
    buffer = malloc(1024);
  }else if(x10 < 123456){
    //infeasible path
    buffer = malloc(x10);
  }else{
    //correct code
    buffer = malloc(1024);
    free(buffer);
  }
  return 0;
}
```

B.4 Program 4

Listing B.4: Program Under Test 4

```
#include <stdio.h>
#include <stdlib.h>
/* This is the demonstration of the violation of
the STRCPY security property */
int main(){
    int x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14, x15,
        x16,x17, x18,x19,x20,x21,x22,x23,x24,x25,x26,x27,x28,
        x29,x30,x31,x32,x33,x34,x35,x36,x37,x38,x39,x40;
    char * buffer;
    char src[]="abcdefghijklmnopqrstuvwxyz";

    scanf("%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%
        d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d",&x1,&x2,&x3,&x4,&x5,&x6,
        &x7,&x8,&x9,&x10,&x11,&x12,&x13,&x14,&x15,&x16,&x17,&x18,
        &x19,&x20,&x21,&x22,&x23,&x24,&x25,&x26,&x27,&x28,&x29,
        &x30,&x31,&x32,&x33,&x34,&x35,&x36,&x37,&x38,&x39,&x40);

    if(x1 > 5 && x2 < 200 && x3 <= 300 && x4 >= 350 && x5 != 400
        && x6 == 5678 && x7 == 87 && x8 == 399 && x9 == 855
        && x10 == 876 && x11 == 234 && x12 == 289){
        char dest[x1];
        strcpy(dest,src);
```

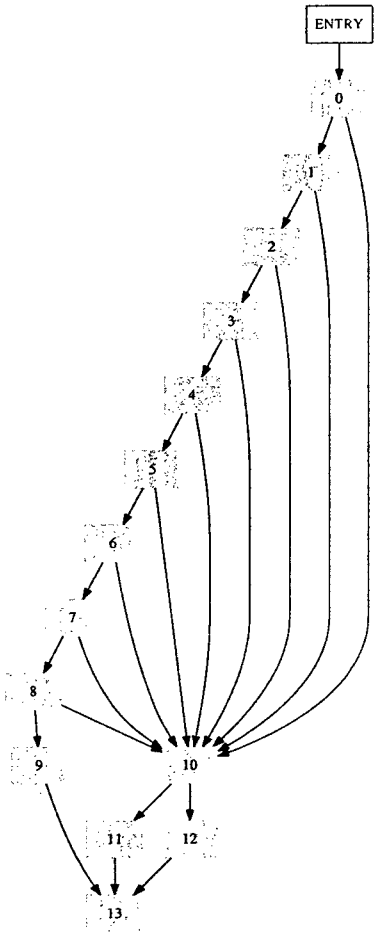


Figure 32: CFG of Program 3

```
    }else if(x13 > 100 && x14 == 111 && x15 == 22 && x16 == 33
        && x17 == 44 && x18 == 55 && x19 == 66 && x20 == 77
        && x21 == 88){
        const int BUFFER_SIZE = 10;
        char str1[BUFFER_SIZE];
        char str2[]="abcdefghijklmn";
        strncpy(str1,str2, BUFFER_SIZE-1);
        /* guarantee result will be null terminated */
        str1[BUFFER_SIZE-1] = '\0';
    }
    return 0;
}
```

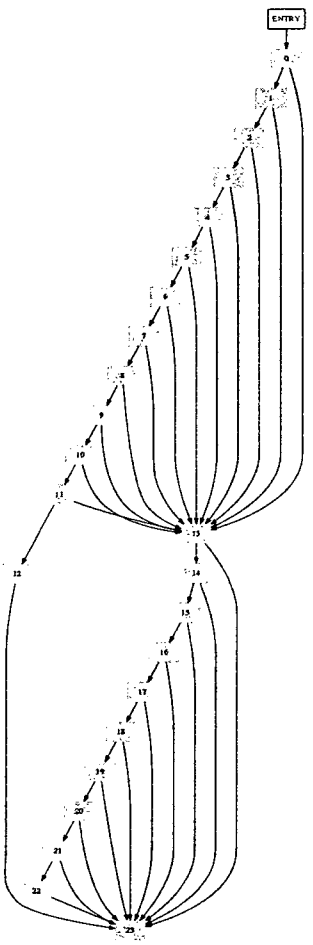


Figure 33: CFG of Program 4

B.5 Program 5

Listing B.5: Program Under Test 5

```
#include <stdio.h>
#include <stdlib.h>

//TMPNAM-TMPFILE security property demonstration
int main(){
    char *filePath;
    int x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,
            x19,x20,x21,x22,x23,x24,x25,x26,x27,x28,x29,x30,x31,x32,x33,
            x34,x35,x36,x37,x38,x39,x40;

    scanf("%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%\
            d%d%d%d%d%d%d%d%d",&x1,&x2,&x3,&x4,&x5,&x6,&x7,&x8,&x9,&x10,&x11,
            &x12,&x13,&x14,&x15,&x16,&x17,&x18,&x19,&x20,&x21,&x22,&x23,&x24,
            &x25,&x26,&x27,&x28,&x29,&x30,&x31,&x32,&x33,&x34,&x35,&x36,&x37,
            &x38,&x39,&x40);

    if(x1 > 1 && x2 > 2 && x3 > 3 && x4 > 4 && x5 > 5 && x6 > 6 && x7 > 7
        && x8 > 8 && x9 > 9 && x10 > 10 && x11 > 11 && x12 > 12 && x13 < -13
        && x14 < -14 && x15 < -15 && x16 < -16 && x17 < -17 && x18 == 18
        && x19 == 19 && x20 == 20 && x21 == 21 && x22 == 1023){
        filePath = tmpnam(NULL);
    }else if(x1 < 1){
    //Examples of Corrected Code
    mode_t old_umask;
    char template[] = "/tmp/fileXXXXXX";
    int fd;
    old_umask = umask (077);
    fd = mkstemp(template);
    umask (old_umask);
    }
    return 0;
}
```

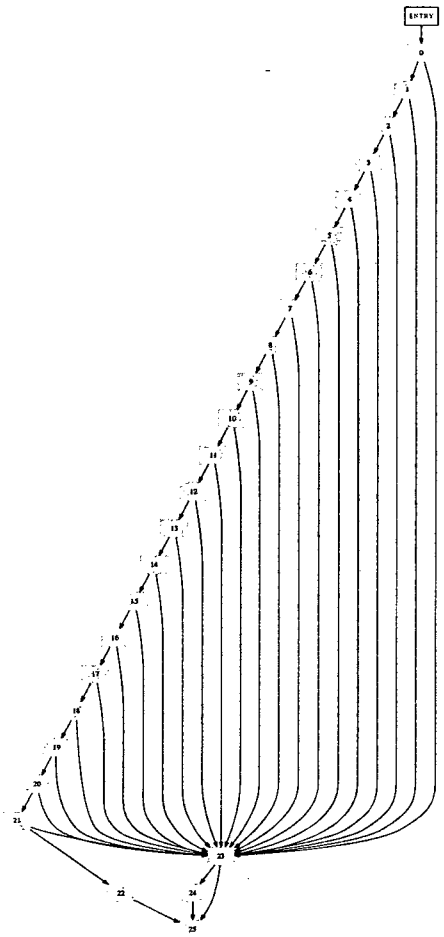


Figure 34: CFG of Program 5