A HYBRID FRAMEWORK FOR THE SYSTEMATIC DETECTION OF SOFTWARE SECURITY VULNERABILITIES IN SOURCE CODE

AIMAN HANNA

A THESIS IN

THE DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

OCTOBER 2012

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: **Aiman Hanna**

Entitled: **A Hybrid Framework for the Systematic Detection of Software Security Vulnerabilities in Source Code**

and submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair
Dr. N. Esmail

______________________________ External Examiner
Dr. A-A. Ghorbani

______________________________ External to Program
Dr. C. Assi

______________________________ Examiner
Dr. S. Mudur

______________________________ Examiner
Dr. D. Goswami

______________________________ Thesis Supervisor
Dr. M. Debbabi

Approved by ______________________________
Dr. V. Haarslev, Graduate Program Director

August 31, 2012

____
Dr. Robin A.L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

A Hybrid Framework for the Systematic Detection of Software Security Vulnerabilities in Source Code

Aiman Hanna, Ph.D.

Concordia University, 2012

In this thesis, we address the problem of detecting vulnerabilities in software where the source code is available, such as free-and-open-source software. In this, we rely on the use of security testing. Either static or dynamic analysis can be used for security testing approaches, yet both analyses have their advantages and drawbacks. In fact, while these analyses are different, they are complementary to each other in many ways. Consequently, approaches that would combine these analyses have the potential of becoming very advantageous to security testing and vulnerability detection. This has motivated the work presented in this thesis.

For the purpose of security testing, security analysts need to specify the security properties that they wish to test software against for security violations. Accordingly, we firstly propose a security model called Team Edit Automata (TEA), which extends security automata. Using TEA, security analysts are capable of precisely specifying the security properties under concerns. Since various code instrumentations are needed at different program points for the purpose of profiling the software behavior at run-time, we secondly propose a code instrumentation profiler. Furthermore, we provide an extension to the GCC compiler to enable such instrumentations. The profiler is based on the pointcut model of Aspect-Oriented Programming (AOP) languages and accordingly it is capable of providing a large set of instrumentation capabilities to the analysts. We particularly explore the capabilities and the current limitations of AOP languages as tools for security testing code instrumentation, and propose extensions to these languages to allow them to be used for such purposes. Thirdly, we explore the potential of static analysis for vulnerability detection and illustrate its applicability and limitations. Fourthly, we propose a framework that reduces security vulnerability detection to a reachability problem. The framework combines three main techniques: static analysis, program slicing, and reachability analysis. This framework mainly targets software applications that are generally categorized as being safety/security critical, and are of relatively small sizes, such as embedded software. Finally, we propose a more comprehensive security testing and test-data generation framework that provides further advantages over the proposed reachability model. This framework combines the power of static and dynamic analyses, and is used to generate concrete data, with which the existence of a vulnerability is proven beyond doubt, hence mitigating major drawbacks of static analysis, namely false positives. We also illustrate the feasibility of the elaborated frameworks by developing case studies for test-data generation and vulnerability detection on various-size software.

To my parents for their endless love,

To Clarice for giving my life a meaning,

To Shana for sharing the good times and the bad ones.

Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Mourad Debbabi, whose expertise and professionalism have added considerably to the completion of this doctoral thesis. His continuous and never-fading support, guidance and insights have been invaluable to me. I am indeed very thankful to have been worked for those many years with such a knowledgeable pioneer in the software security field.

I would also like to extend my sincere thanks to my fellow researchers, the members of the TFOSS team. Special thanks to Dr. Dima Al-Hadidi, Dr. Nadia Belblidia, Dr. Rachid Hadjidj, Mr. Terry Zhenrong Yang, Mr. Eric Hai Zhou Ling, Mr. Xiaochun Yang, Dr. Chamseddine Talhi, Mr. Andrei Soeanu, and Dr. Yosr Jarraya. I have been quite fortunate to work with every and each one of them.

Finally, but by no means least, I would like to express all my thanks and my deepest appreciation to all my family members. Their tremendous support through those many years, from which some have been quite difficult, is the reason I was able to carry on. Special thanks to my mother, Mrs. Hoda Raphael, my brother, Mr. Ashraf Hanna, my best friend, Shana, and my little angel, Clarice. This thesis is given in memory of my father, Mr. Latif Hanna, who left us early this year but will always be with us.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we explain the motivations behind the research work presented in this thesis, as well as our objectives, approaches, and contributions. Section 1.1 provides a brief background of the main different methodologies for securing software. Section 1.2 provides a brief introduction to Free-and-Open-Source-Software (FOSS). Section 1.3 highlights the motivations and the scope of the problem addressed in this thesis. Section 1.4 details the objective of our research work. Section 1.5 highlights the contributions of our work, together with references to the various publications resulted from this research work. Finally, Section 1.6 highlights the structure and organization of this thesis.

1.1 Software Security Methodologies

The past few years have witnessed an exponential growth of software utilization. The movement from strictly single-function hardware, not long ago, to a hardware that can behave in dramatically different ways through the use of software is rather apparent and indeed enjoyed by many of us. Today, software represents in one way or another a major part of most, if not all, human beings lives. It is quite difficult to imagine any major industry today that does not take advantage of software. Whether it is medical imaging, online-shopping, banking, education, cellular networking, satellite communications, etc., software has managed to manifest itself into such industries. In fact, software turn to represent a rather significant component of many of such industries.

Yet, the increased utilization of software in the past decades has not been met with an equivalent, or even relative, growth of concerns in relation to software security. As a matter of reality, software attacks still make first-page news headlines around the world. As an example, on February 5, 2010, BBC News reported that Microsoft is to patch a 17-year-old vulnerability. The vulnerability dates back from the DOS operating system and has then been carried over into almost every version of Windows that has appeared since [15]. In fact, as concurrently reported by BBC, Microsoft was to tackle a further 25 loopholes in Windows at the same time, five of which are rated as "critical", and that this update is not even the largest that Microsoft has ever released. On October 2009, Microsoft has tackled a total of 34 vulnerabilities, where 8 of those were rated as critical, the highest level [15]. Such security vulnerabilities and their consequent attacks represent a major challenge to the software industry, where end-users expect security to be delivered out of the box. Yet, most programs are not written by security analysts, and in many cases are not even designed with security in mind.

The vitality of the problem at hand has resulted in various security research efforts being conducted in academic, corporate and governmental organizations. The most prominent proposals could be classified as: security engineering, security hardening, and security testing.

Security engineering [11] attempts to consider security at the early stages of the software development cycle. Although this approach, if followed, may indeed lead to more secure software, its applicability in reality is questionable. Software developers often face strict deadlines due to various market pressures, and hence they usually strive to produce working software without much regard of other aspects, including security. Another problem with the security engineering approach is that a great deal of software has already been developed and deployed. In these cases re-engineering is needed and could still be possible. However, there is an attached cost to re-engineering that cannot be overlooked. In fact the cost estimates of re-engineering a system are often compared to the cost of replacing or re-developing that system [146]. Such cost may turn the re-engineering approach unattractive in many cases.

Security hardening [115] is an attractive approach where hardened security routines can be injected into existing code to produce more secure software. However, security hardening requires the work to be done by a developer or by a personnel with sufficient programming and security background. Additionally, in order to harden software, the engineer must first know where the faults are, and so inject the hardening routines at the appropriate locations. In general, we view security hardening research as a very interesting subject that particularly complements software security testing.

While security engineering and security hardening are relatively precise to define, security testing is a much broader field. Security testing taxonomy encompasses various fields such as security code inspection, vulnerability scanning, security auditing, penetration testing, and others.

Various research efforts have targeted security testing in the past few years. The scope of concern and detection capabilities of the proposed approaches differ. For instance, some of the approaches involve a reasonable amount of manual intervention either by developers or by code-reviewers to reveal vulnerable code segments in the software. However, this is usually quite difficult to apply, especially for large-size software, which limits the applicability of such approaches. Other proposals have compiled common errors and vulnerabilities in code production languages such as C/C++. The intent is to make aware software developers of what, and what not, to do when writing software. While such proposals are good and could result in better code being written, developers are not bound to follow such guidelines. Consequently, developers often discard such guidelines either fully or partially. Validating the software against such guidelines is possible but also time-consuming, which limits its applicability in reality.

Research efforts, such as [28, 63], have proposed more comprehensive approaches. However, these approaches have specific testing focus. This, along with the underlying methodologies of these approaches, have direct impact on their detection capabilities and limitations, as we explain in Chapter 2.

1.2 Free-and-Open-Source-Software (FOSS)

Free-and-Open-Source-Software (FOSS) is software that is liberally licensed to grant the right to users/developers to access the software and use/modify it in any way they wish to. While commercial software has initially dominated the software market, the past few years have witnessed a vast shift towards the dependence on free-and-open-source software. The high cost of commercial software, including multiple after-purchase costs, such as upgrades and support contracts, and its closed-source code nature are among the causes of such shift. In contrast, open-source software provides software developers with a great flexibility by permitting reading, changing, enhancing and maintaining the source code. Today, the utilization of open-source software is on the rise, by both individuals and organizations. For example, in 2002, a research financed by the European Commission under the Information Society Technologies (IST) Thematic Programme, showed that 395 out of 1452 (about 27%) large-size organizations in Germany, Sweden, and the UK were indeed using, or planning to use within one year, some sort of open-source software [17]. A Netcraft's survey [121] published in April 2007 polled all the websites they could find (totaling $113,658,468$ sites), and found that Apache had 58.86% of the web servers market, while Microsoft had 31.13% [160]. In 2006, an IBM-sponsored study [19] suggested that GNU/Linux has "won the server war" as 83% were using GNU/Linux to deploy new systems.

1.3 Motivations and Problem Statement

The problem being addressed in this thesis can precisely be described as the detection of security vulnerabilities, by security testing, in free-and-open-source software, or generally in software where source code is accessible. In this context, we examine multiple major issues related to this problem, including the detection methodology, the precise capture of security requirements, and the detection techniques. Consequently, matters such as the specification of security properties, code instrumentation and test-data generation are explored. Moreover, a special care is dedicated to the level of automation, detection power, reduction of false-positive reporting, and others.

Various reasons compose the rationale behind our choice to address this particular problem. Firstly, software security problems are very much a matter of our day-to-day reality. Some of these problems are severe enough to make first-page news headlines. A Secunia report [58], in the first quarter in 2010, has shown that the number of reported software vulnerabilities has not declined in spite of all efforts in the past few years to lessen the software security problem. As a matter of fact, the number of reported vulnerabilities in 2009 has roughly increased by nearly 20% in comparison to 2005 [58].

Secondly, problems arising from security vulnerabilities have an overwhelming financial impact that can be greatly alleviated through security software testing. It was estimated that software failures currently cost the US economy alone about $\$60$ billion every year [31], and that improvements in software testing infrastructure might save one-third of this cost.

Thirdly, an attractive solution to the vulnerable software problem would be to test the software before deploying it, so that it carries no, or even less, security vulnerabilities. While such solution seems ideal, we understand the reasons behind the lack of its applicability today, as manual intervention is required. Actually, security testing is a non-trivial process. Conventional testing, as proposed in many software engineering books and guidelines, is often both time-consuming and financially costly. For instance, in order to increase the chances that a software unit is error-free, full-path coverage may be required using unit testing. However this is usually expensive in terms of time and hence cost. Consequently unit testing is either poorly conducted or simply discarded altogether. Hence, automation of the testing process is rather crucial.

Fourthly, we realize that having access to the source code of a software can be a great asset to lessen security problems. Realistically, source code is not always accessible. Instances of such include commercial software, and software where the source code is lost or is unavailable for different reasons such as licensing issues. However, in other cases, such as free-and-open-source software, the source code is accessible. In such cases, it is possible to conduct security testing over the source code to enhance software security.

Additionally, with the increase in utilization of FOSS today, as well as its more susceptibility to attacks, since the source code is available to attackers as well, efforts in relation to detecting vulnerabilities in FOSS would be very advantageous to the field of software security.

Finally, another important point is that further work is still needed today in relation to security. While various research efforts have indeed targeted different security problems, the focuses of such works differ from our focus, as we detail in Chapter 2. Consequently, contributions that address the problem targeted in this thesis are needed.

The above reasons/facts have strongly driven us to conclude that contributions to solve the aforementioned problem are quite significant to the software security field today, and it is worthwhile spending efforts to contribute to solving it.

1.4 Objectives

The primary intent of this thesis is to elaborate a methodology and a framework for the automation of source-code (particularly FOSS) security testing. More specifically, our objectives are to:

- Assess the available approaches for software security testing, identify their strengths/shortcomings and extend their relevant methodologies to build a framework for systematic security testing.
- Define an expressive language for the specification of both high-level and low-level security properties.
- Define a practical and efficient framework for code instrumentation. Additionally, leverage the aspect-oriented programming paradigm to define security-dedicated primitives that enable expressive instrumentation.
- Elaborate efficient techniques for test-data generation.
- Design and implement a software environment for automating security testing and hence vulnerability detection, together with demonstrating case studies.

1.5 Contributions

This section highlights the contributions of the research work presented in this thesis, together with references to the different publications that resulted from this work.

The contributions can be summarized as follows:

- Defining an expressive specification language for both high-level and low-level security properties [70, 162];
- Leveraging the aspect-oriented programming paradigm to define security-dedicated primitives for expressive security instrumentation [16];
- Proposing and designing a novel code instrumentation approach based on compiler-assisted and AOP pointcut model. Additionally, extending the GCC compiler for code instrumentation [70, 71, 162];
- Proposing and designing a reachability analysis static-based approaches and framework for vulnerability detection and test-data generation [9, 129];
- Proposing and designing a generic hybrid dynamic-static approaches and framework for automating security testing and test-data generation [71, 98, 129].

The following sections describe the research issues and results that are addressed and achieved in this thesis.

1.5.1 Security Model - Team Edit Automata

In order to test software against particular security properties, analysts first need to accurately specify such properties. One of the main contributions of this work is designing a language with which security analysts are capable of specifying the security properties that they wish to test the software against. The language is designed with specific factors in mind, such as the capability to specify a very large set of security properties, and ease of use. The related contributions are:

- Proposing a language that enables security analysts to accurately, yet easily, specify the security properties in concern. Beside having some basic knowledge of automata, security analysts do not need to have any further expertise to utilize the language.
- Elaborating a security automata model, namely Team Edit Automata (TEA), that provides an enhancement over existing similar models. This model combines the capabilities of previous models and provides an enhancement over their expressiveness, as we detail in Chapter 3.
- Design and implement an integrated development environment that encompasses a graphical user interface for specifying security properties. Such interface facilitates the specification of the properties. Instead of requiring the analysts to provide a concrete code to specify the property, they only need to graphically draw the desired automata. Our model then transforms such graphical automata into the needed concrete code.
- Provide a suite of specifications for common security properties. With such suite, the analysts do not need to specify the automata targeting these known properties; rather just use the provided ones as long as the verified property is application independent. If the property to verify is application-dependent then the analyst has to specify it. This suite can further be extended by the analysts.

1.5.2 Code Instrumentation Unit

In order to achieve various functionalities at both static and dynamic stages, there is a need to instrument various code segments at different program units. For instance, in order to collect behavioral information of the program at runtime, profiling code segments are to be injected at specific locations. For the purpose of achieving automation, there is a need to suppress and substitute particular method calls. The primary contribution of this work is the design and implementation of a code instrumentation API that enables such needed instrumentations to take place. This code instrumenter is used to inject monitoring code at specific locations in the program depending on what to be achieved. This unit is particularly crucial to the dynamic analysis functions of our model, where profiling information is needed for different purposes such as vulnerability detection and test-data generation. The related contributions are as follows:

- Design a code instrumentation unit that enables the injection of profiling/monitoring code at various desired points in a source code.
- Elaborate on needed extensions to AOP, which would enable the language to be much more usable and appropriate for security testing.
- Incorporate the code instrumentation capabilities of compilers with the AOP pointcut model to provide an effective code instrumentation technique.
- Provide a compiler that allows the needed code instrumentations; in specific, extend the GCC compiler for such purpose.

1.5.3 Automatic Test-Data Generation

For the purpose of proving that a specific vulnerability indeed exists in a software, it is necessary to generate test-data, with which program execution would lead to this vulnerability being exploited. The primary contribution of this work is a test-data generator, with which a suspected existence of a vulnerability can be proven beyond doubt; that is eliminating false-positive reports. This generator aims at generating test-data that would exploit a targeted vulnerability. The generator differs from other known data generators in its novel nature, as being goal-path oriented. The generated data would result in specific program points being reached through the execution of specific paths. The related contributions are as follows:

- Elaborate an approach that enables automatic generation of test-data.
- Demonstrate the applicability of mere dependence on static analysis for test-data generation.
- Propose a hybrid dynamic-static analysis approach for test-data generation.
- Elaborate a security vulnerability detection model for automating security testing.
- Analyze the proposed approaches and methodologies against the detection of security vulnerabilities with real-life security testing case studies.

1.6 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the current literature on the subjects that are related to the problems addressed in this thesis, together with the most relevant research proposals. In Chapter 3, we present a language with which security analysts are capable of specifying the security properties. In Chapter 4, we present our approach to code instrumentation. In Chapter 5, we present our mere static-analysis based approach to test-data generation. In Chapter 6, we illustrate our generic hybrid static-dynamic approach to test-data generation. In Chapter 7, we provide the details of the conducted experiments over our models, together with supportive results. Concluding remarks as well as a discussion of future work are reported in Chapter 8.

Chapter 2

Background

In this chapter, we present an overview of the current literature on the subjects that are related to the research problems addressed in this thesis. In this context, we initially present a general view of software security, and differ it from other fields of IT security. We highlight the main different approaches to securing software, and underline their advantages and limitations. A background on two major subjects, code instrumentation and test-data generations, is then presented. The rest of the chapter is organized as follows. In Section 2.1, we present a general overview of information and computer security. Section 2.2 provides a particular classification of software vulnerabilities, as being low-level or high-level vulnerabilities. Section 2.3 provides a background of software testing. Section 2.4 provides a background of software security testing techniques and detections. Section 2.5 provides a background of code instrumentation. Section 2.6.1 provides a background of test-data generation. Finally, in Section 2.7, we provide concluding remarks about what is covered in this chapter.

2.1 Information, Application and Software Security

With the exponential increase of software utilization in the past few years, security has become a critical concern. However, in this context, the term security actually implies a large spectrum of fields. While these fields are actually interrelated, they are distinct.
According to Title 44 of the United States Code [155], the term information security refers to the subject of protecting information and information system from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide integrity, confidentiality and availability. It is hence clear that the term is not restricted to computer security since it applies to other fields, such as printed data. Further, computer security may very well focus on other aspects, such as ensuring the availability of a computer system without any regard to the correctness or the availability of the information stored/used by the computer. Consequently, the approaches and methodologies used to provide security for each of these fields differ.

On another hand, application security encompasses the use of software, hardware, and procedures to protect applications from various threats. Generally, it refers to the matter of how a software application should generally behave, so that it complies with a defined security policy. In theory, in order to implement a secure application, all needed measures must be considered during the development life-cycle of the application, so that the implemented application is secure. While in many cases very strict measures are actually put in place during the development of the application, it is almost often that the deployed application may carry various vulnerabilities. There are many reasons behind this reality including market pressures to produce a functional software within a bounded amount of time and the fact that many parts of the applications are usually written by software engineers who are not security specialists. Nonetheless, it is clear that software security is just a particular filed of application security, which is not limited only to software. Problems related to software security are mainly driven by coding defects that result in the software becoming vulnerable to attacks. However, in the context of this thesis, we tend to follow a more precise definition of an insecure/vulnerable software, as follows: A software is vulnerable if its behavior violates a given security property. This definition provides the scope and focus of software security considered in this research. In the following sections, we provide a classification of software vulnerabilities and highlight the major methodologies for securing software.

2.2 Low-Level and High-Level Security Vulnerabilities

In computer security, the word vulnerability refers to a weakness in a system, with which, if exploited, an attacker may become capable of violating one or more security properties of the system, such as confidentiality, integrity, and availability. In general, a vulnerability may be viewed as an entity that exists only in theory, until the moment when it is exploited. It can also be viewed as an entity that has a significant interest to attackers, especially when the program containing it operates with special privileges, such as providing access to user/firm data or other resources, such as network servers. The source of vulnerability may vary, although often it is initiated by a programmer's carelessness. For example, a specific vulnerability may be initiated during design phases. Such design flaw may naturally result in the vulnerability being propagated through the final implementation of the software.

There are many ways to categorize vulnerabilities. For example, we can view vulnerabilities based on their causes. For instance, some vulnerabilities are caused by user inputs. A PHP/HTML/JavaScript web page, for example, that allows a user to enter data, to perform search queries, may allow an attacker to corrupt the site's data and permanently damages the behavior of this page. This can be done by the attacker entering simple HTML controls, such as "$< script >$", as input. SQL Injection is another example of such cases, where the user's input leads to improper or unauthorized execution of SQL statements. Some other vulnerabilities are caused by programmer's failure to check the size of a data buffer, hence leading to an overflow. Consequently, this may cause corruption of some memory areas including the stack and the heap and allows the system to execute code provided by the attackers.

It should be noted that there is a very thin line between the two causes just described. For example, it is reasonably valid to also blame the programmer for failing to validate and correct the user's input in the first example; for instance by mapping the input through some functions such as *htmlentities*, *addslashes*, etc.

We here follow a particular classification of security vulnerabilities, which is according to the security properties they violate. Security properties can also be categorized as high-level security, or low-level security. We further refer to the former as security properties and to the latter as safety properties.

High-Level Security

There are mainly three high-level security properties, namely integrity, confidentiality, and availability. We here overview these properties, and then highlight some of the techniques and mechanisms needed to enforce these properties.

Integrity: Integrity is the process of ensuring that information will not be accidently or maliciously altered or destroyed. Often, information go through various processing operations, such as transferring, storing, forwarding, and retrieval. Data integrity is the process of ensuring that no unauthorized or accidental alteration has been applied to it during processing.

Confidentiality: Information confidentiality is the process of ensuring that there is no leak of information to unauthorized parties. Since data confidentiality represents one of the most significant concerns today in relation to information security, a stricter definition of it, following the military Need-to-Know principle [3] may be more appropriate. Simply put, information that is considered to be confidential in nature must only be accessed, used, copied, or disclosed by entities who have been authorized to access, use, copy, or disclose the information. Moreover, this can only be done when there is a genuine need to access, use, copy or disclose the information.

Availability: Data availability is the process of ensuring that data, resources to access this data, as well as security controls over this data and the recourses, are all available and functioning as expected when access to this data is required. The unavailability of any of these is best known as Denial of Service (DoS).

In order to ensure these high-level security properties, some techniques and mechanisms are needed. Below, we provide examples of such mechanisms.

Authentication: In software, authentication is the process of determining whether a user/process is, in fact, who or what it is declared to be. Today, the most common method for authentication, in both private and public networks, is through the use of users' identifications and passwords. Providing the correct password is assumed to guarantee that the user is authentic. This method however is too weak, especially for transactions that are highly sensitive, such as military or financial transactions. Passwords can often be stolen, guessed, or accidentally revealed. For example, it is not difficult to mimic the look of a financial institution webpage, and redirect users to it instead of to the appropriate site for the purpose of collecting users' passwords. The redirection can also be done easily. For instance, many users keep the web addresses of their financial institutions as a favorite link in their browsers. An attacking process can easily manipulate the Uniform Resource Locators (URLs) behind these favorite links. Further use of these favorite links would then redirect the user to the attacker's site instead. Another serious issue with passwords, which is often totally overlooked, is that although passwords are indeed secrets that are chosen by the appropriate person/entity, they are finally stored in databases, which can be accessed by others. This, in fact, has motivated the creation of more stringent authentication schemes, such as digital signatures.

Privilege Escalation: Privilege escalation is the act of wrongly allowing an entity (user, process, etc.) to gain access to resources, which normally would have been protected from this entity. In other words, that is allowing an entity to execute actions with a higher security context than what it is supposed to. Privilege escalation often occurs as a result of a bug exploitation, or when an application with high privileges has flawed assumptions about how it will be used. For example, a high-privilege application may assume that it will only be provided with input that matches its interface specification, and hence does not validate the input. An attacker may then be able to exploit this assumption so that unauthorized code is run with the application's privileges. For instance, in certain versions of Linux kernel, such as Linux 2.6.13, it is possible to write a program that would request a core dump be performed in case it crashes at runtime. An attacker program would simply set its current directory to `/etc/cron.d`, then has itself killed by another process. Consequently, upon the termination of the process, the core dump file is placed under `/etc/cron.d`, and `cron` would treat this file as a text file instructing it to run programs on schedule. Because the contents of the file is under attacker's control, the attacker would then be able to execute any program with root privileges [74]. Cross-zone scripting is another form of privilege escalation attacks. The security zone concept classifies websites into different zones according to the limits of what a website should be able to do. Consequently, a web browser would view any site as a part of one zone or another. For example, Internet Explorer 4, and later versions, divide all sites into 4 zones [112]: Local Intranet, Trusted Sites, Restricted Sites and Internet (everything that does not belong to one of the other zones). The first 3 zones are usually configured as privileged zones. Cross-zone scripting attacks sabotages the browsers security model, as a result of a browser's bug or a configuration error, so that a web site can run malicious code on the client's machine.

Authorization: Authorization refers to the restriction measures to resource use. In respect to the authorization model, software entities are viewed as subjects and objects, where subjects are the entities that can perform actions in the system and objects are the entities on which actions are performed and possibly need to be controlled. In that context, authorization is the process that decides whether or not a subject is allowed to access an object.

Non-Repudiation: The term non-repudiation is possibly more traditionally common in the legal system than in information technology. The definition of the term in both fields is the assurance that contract terms will be carried by the parties involved and that the identity of these parties is validated/assured to be the correct identities. In information technology, non-repudiation can alternatively be defined as the impossibility for one of the entities involved in a communication denying having participated in all or part of the communication. For example, if a sender and a receiver agreed to send and receive information respectively, then the transmission and receiving of information must be carried. Hence, a non-repudiation of transmission proves that the data has been sent and a non-repudiation of receiving proves that the data has been received. It is clear however that means of repudiation in the legal system significantly differ from those in information technology.

It should be clearly noted that the above-mentioned mechanisms for enuring high-level properties are not inclusive. Other important mechanisms include atomicity, fairness, certification, secrecy, and revocation. Additionally, various significant security mechanisms and technologies exist for the purpose of achieving some of these security properties. For instance, the Advanced Encryption Standard (AES) and Rijndael Algorithm [44], the Rivest-Shamir-Adleman (RSA) Algorithm for encryption [133], Digital Signature Standard (DSS) for verification [156], the Clipper Chip and Skipjack Algorithm for encryption and privacy [52], and Pretty Good Privacy (PGP) for encryption of e-mails over the Internet [128], are among such mechanisms. It should be noted that the available mechanisms may not be sufficient to achieve the full spectrum of security requirements. Moreover, some of these mechanisms are themselves subject to attacks and possible depreciations. For instance, RSA-576, RSA-640, and RSA-768 were factored on December 3, 2003, November 2, 2005, and December 12, 2009 respectively [137].

Low-Level Security - Safety

During the past few decades various programming languages have been introduced, from which some have gained an overwhelming popularity. There is no doubt that the successes of these popular ones were directly related to their level of capabilities and power in producing more complex software. It might be quite reasonable to state that C, and its successor C++, have enjoyed the very top of the popular list for quite a while. As a matter of reality too, a massive amount of software has been, and still being, developed using them. Unfortunately though, C and C++ have also led another list as being among the most vulnerable languages in terms of security. It has been argued that the maximal performance and power of these languages has been achieved at the cost of the security level that they provide.

The relation between low-level security and the used language is direct. Low-level security is extremely dependent on the programming language, or in other words, on loopholes that a programming language would allow. However, programming languages are not the only reason behind low-level security vulnerabilities. For example, low-level security vulnerabilities may be caused as a result of design or implementation errors in a specific platforms, architecture, or system software, such as operating systems. There are various low-level security vulnerabilities. In the sequel, we list and discuss some of the major ones.

Buffer Overflow: Buffer overflow occurs as a result of allowing more information to be stored in a bounded buffer, and above the capacity of what this buffer is capable of holding; consequently allowing illegal access to other areas in memory. In some fortunate cases, attempts to access illegal memory locations may just result in memory access exception and program termination. In worst cases, a malicious user may use buffer overflow to breach system security. Unfortunately, it is often quite easy, especially with some programming languages, for attackers to stress on this vulnerability to breach system security in one way or another. Such attacks may result in crashing the program or making it behave in an unintended way. For instance, the 1997 Cisco *7xx* routers password overflow [33] is an example of how easy, and how serious, a buffer overflow problem can be. In this incident, an attacker needs only to `telnet` to the router, or to gain access to its console port, so that he/she is allowed to enter a username/password. It was only sufficient to crash the router by entering a very long password, which overflowed the buffer allocated to read the password. Consequently, the routers crashed, breaking the network paths and resulting in a denial of service attack. Worst, as reported by Cisco, other possibilities did exist where the attacker could exploit this problem to launch different attacks against the router, rather than just crashing it [33]. By including the right data at the right place in the too-long password string, an attacker could cause the router to hang indefinitely, or to take complete control of it, including reconfiguring it, or modifying its functionality, theoretically in any way at all.

Although buffer overflow may be most known as memory-related overflow, we should emphasis that the types of memory being exploited vary. We here briefly describe some of these various types of exploitation. We also discuss some of the methodologies and techniques used to remedy them.

Heap Overflow: This is also known as heap-based exploitation. The heap is a memory area that is dynamically allocated to a program at runtime, and usually contains program data. An attacker can stress on heap overflow to change the content of the heap, possibly causing the application to overwrite some of the dynamically allocated internal structures, such as pointers and linked lists. The 2004 Microsoft JPEG vulnerability is an example of such overflow [111], where a heap overflow could be exploited as a result of the processing of a JPEG image by some Microsoft software components. An attacker who could successfully exploit this vulnerability would gain complete control of the affected system. With this vulnerability, the attacker just needed to persuade a user to open a specially crafted image file, or to view a directory that contains that file; this could also be possible through persuading the user to connect to a web page that hosts the image.

Stack Overflow: This is also known as stack-based exploitation. A stack, or more accurately a call stack, is a bounded memory buffer that keeps information about active subroutines. Stack buffer overflow occurs when a program writes more data to a buffer located in the stack in excess to the capacity of that buffer. Consequently, this almost always results in the corruption of adjacent data of the stack. An unintentional occurrence of stack overflow will often result in program crash or misbehavior. On the other hand, a malicious user can use this problem in various ways to severely harm the system. For example, by overflowing the buffer in the stack, the user may be able to change a nearby value of the stack, such as a subroutine return address, a function pointer or an exception handler code. A change of a subroutine return address, for instance, could allow the attacker to point the program once returned to a different address, and hence execute a different code, which is usually provided by the attacker.

Integer Overflow: Many programming languages introduce various integer types, and permit conversions between these types; either explicitly or implicitly. For examples, compilers do distinguish between unsigned and signed integers, and allow various operations on them, such as implicit casting, integer truncation, integer rounding, overflows and underflows. Some of these operations, if overlooked, can turn very severe. Generally, integer overflow security issues arise either by such conversions, or as a result of their inherently limited range [143]. These problems can be remedied however if integer operations are replaced by safer code that performs range checking before carrying out sensitive operations [143].

File Management: File manipulations, such as creation, reading, writing or deletion, are allowed by many programming languages. Although file manipulations are needed, a misuse of these capabilities may result in very sever security issues, such as data corruption, data disclosure, code injection, illegal privilege escalation, and others. For example, a common practice is the use of temporary files to support the execution of an application. These files are possibly created with predictable names and without prior verifications that files with the same names do not already exist in the temporary file directory. A local attacker may take advantage of this by creating symbolic links to the temporary files used by the vulnerable application. When a target user runs the application, the temporary files will be overwritten with the rights of this user. This makes it possible for the attacker to then create or overwrite existing files in the filesystem with the elevated privileges of the target user [144].

Memory Management: We could reasonably state that the most powerful, yet very dangerous, operator in any programming language is the pointer operator "*" introduced by C/C++. These languages allow a great deal of expressiveness power to programmers since they can directly point, or attempt to point, to virtually any memory location. Although some level of protection is provided, for example by operating systems, to prevent such illegal access of memory, many other memory management problems are still possible, and usually provide attackers with a great help. Some of the very common pointer manipulations might lead to memory leak, dangling pointers, double-free, use of un-initialized memory, and use of previously freed memory. Better programming practices and proper use of pointers might provide the best remedies to these problems. However, since this is not a practical assumption, other solutions such as software hardening, or use of safer APIs is needed to solve these problems.

2.3 Software Testing

With the tremendous growth of software size and complexity in the past two decades, software testing has become a very challenging subject. Particularly, matters such automation and detection powers have turned to be significantly important. Various researches have consequently taken place in order to provide an acceptable software testing approaches and routines. Below we describe some of the initiatives that have taken place in relation to the subject. The main reason behind our review of these research initiatives is due to the fact that we take advantage of some of the techniques described or used in these works. However, it must be noted that these approaches target software testing in its general form, which are not specific to security testing and vulnerability detection. For instance, the work described in [14] is very useful for testing under extreme cases, where the value of a state variable is at either maximum or minimum of its sub-domain. This however does not target security testing in particular, where vulnerabilities may occur while the state variables are far below, or above, maxima or minima. The work described in [6] is very useful for testing large software, since the approach is to break it into smaller units. However, the approach targets the detection of program faults such as crashes and assertion violations, which may not be related at all to testing against particular security violations.

Section 2.3.1 provides a background and the related work in relation to subject of formal methods and formal specifications. Section 2.3.2 discusses the research works in relation to software testing techniques and automation. Section 2.3.3 provides the related work in relation to security policy specification. Section 2.3.4 discusses the relation between formal verification and testing and the related work to this subject.

2.3.1 Formal Methods and Specifications

The matter of formal specification may not at a first glance seem related to testing. However, this is inaccurate since the use of formal methods to specify the software functionality and behavior at early stages tend to indeed produce higher quality software. This leads to a reduction in cost and time in discovering and fixing system misbehavior. Formal specification languages are designed to assist with the construction of systems, including software. Below, we provide a background on the main formal specification languages.

Specification Languages

The main goal behind the utilization of formal languages, in relation to software, is to allow for the writing of precise specification of the software. Both textual and graphical notations can be used to achieve what is needed. This specification can additionally target both the functional behavior of the system, as well as its non-functional aspects, such as security. In that regard, some of the proposed specification languages target the general aspects of a system, while other are proposed with specific target scope in mind. We here provide a background on the main categories of formal specification languages.

Finite State-Based Languages

The main idea behind finite state-based languages is the representation of system states as states of a finite transition system; i.e. state machines. The representation of such state machines can be done either through textual or graphical representations. Examples of utilization of such approaches include Finite-State Machines (FSMs) by Lee et al. [39], X-machines by Holcombe et al. [105], and Statecharts by Harel et al. [38]. Generally, testing based on FSM-based languages is capable of naturally achieving matters such as conformance testing. However, FSM-based testing is not limited to this. For instance, they can be combined with other approaches to produce more powerful testing approaches. For instance, it is possible to first construct a model of the system, for instance using model-checking, then drive the testing process based on these constructed models. Examples of such approaches have been illustrated by Grieskamp et al. [158], and Farchi et al l. [46]. A general definition of a FSM, *F*, can be given as follows:

$\langle Q, Q_0, I, O, \delta, DF \rangle$, where:

- Q is a finite set of states,
- Q_0 is a finite set of initial states such that $Q_0 \subseteq Q$,
- I is a finite set of input symbols,
- O is a finite set of output symbols,
- $DF \subseteq Q_0 \times I$ is the specification domain, and

- $\delta : DF \rightarrow \mathcal{P}(Q \times O)$ is a transition function, where $\mathcal{P}(Q \times O)$ is the powerset of $Q \times O$.

Now, given a FSM *F*, it is possible to obtain a corresponding language *L*, such that *L(F)* defines all the possible sequences of *F* starting from any possible input at the initial state. Such sequences can then be used very effectively for testing purposes by tracing the behavior of a system represented by such a machine. Additionally, many specification languages that deploy a finite state structure also incorporate additional internal data. In such cases, transitions can represent operations that manipulate such data, possibly through guards that allow/disallow such manipulations. These machines are referred to as Extended Finite State Machines (EFSMs).

Model-Based Languages

Precise specification of a system can be achieved through the construction of a model that corresponds to the intended behavior of the system. Such a model can be described in various ways. For instance, it is possible to describe the states of the system together with operations that performs changes to these states. Specifications languages such as the Z language [79, 80], the Vienna Development Method (VDM) [23] and the B Method [82] can be used to allow for such precise specifications.

Such languages can describe arbitrarily systems with possibly infinite set of states. While this may be attractive in some cases, it is considered as a shortcoming when automation is required due to the difficulty to automatically reason on the potentially infinite behavior of such systems. It is worth noting that FSM-based languages do not suffer such shortcomings.

2.3.2 Testing Techniques and Automation

Software testing aims at uncovering faults in software. Such faults may result from early incorrect specifications or from incorrect design or implementation. Traditionally, the software testing process is described as a sequence of testing phases. These phases are initiated as unit testing, where individual units are tested in isolation. This is followed by integration testing, where multiple components are combined and testing is then performed over these components. This later process repeats until all system components are put together and the system is them tested as a unit, which is referred to as system testing. Acceptance testing is then conducted to find out whether the system specification and requirements are met by the implemented system. While the testing process can manually be conducted in some cases, the increased of software complexity and sizes today may turn this task as a next to impossible one. Consequently, automation of software testing is rather crucial. The importance of such subject has resulted in various research efforts.

While it is clear that automation is needed to efficiently conduct software testing today, automation is a challenging subject that has its own obstacles. Different techniques have been proposed to overcome different problems. Below, we provide a background on some of the main techniques in relation to the subject.

Software Partitioning for Testing Automation

One of the main obstacles when it comes to software testing is the size of the software. The idea of software partitioning was proposed to mitigate such problems. In [134], Hierons introduced an algorithm for partitioning software based on their input domain. The algorithm required the software specification to be expressed in the Z language. The generated partitions can be used for the generation of test cases as well as to produce a finite automata model, which can be used to control the testing process. The main idea was to rewrite the specification to an intermediate form used to derive both partitioning of the input domain and the states of a finite state automaton model. Test cases are then derived from the partitions, while an automated system that controls the testing process is based upon the finite automaton. In [138], Burton used user-defined testing criteria, expressed as Z specifications, to automatically generate test cases. To facilitate automation, formal specification of the heuristics for generating the tests was required. The specification also allows for desirable properties to be checked and for a comparative analysis to be carried out between testing criteria.

In [14], Legeard et al. proposed two methods for automatic generation of tests, namely the Test Template Framework (TTF) and the B Testing Tools (BTT). The first requires a specification given in Z language and the manual generation of test cases, while the later works over B specification, and uses constraint logic programming techniques to generate test sequences. The proposed methods targeted boundary testing, and avoided the construction of a complete finite state hence mitigating the state explosion problem.

In [25], Meudec proposed a language, VDM-SL, used for expressing the specifications then for deriving efficient test sets. VDM-SL extended the idea of partitioning and TTF to work better with qualifiers. An interesting conclusion by Meudec pointed out that constraint logic programming seems promising, yet a complete automation may not be that achievable.

In [6], Chakrabarti et al. proposed partitioning to effectively automate unit testing. The idea was to identify the control and data inter-dependencies between the software components using static analysis of the program. Based on these analyses, the software is then divided into units, where highly-related components are grouped together. These components can then be tested in isolation using automated test-generation tools. In [141], Spacey et al. examined the speed factor in relation to software partitioning. They demonstrated that a faster execution can be obtained if the partitioned code segments are instantiated on more than one location. Further, they demonstrated that the run-time of the program is not only dependent on the frequency of calls to these segments but also on the order on which these calls are made.

Other Testing Automation Techniques

Other techniques for testing automation have been proposed. For instance, in [109], Vaziri-Farahani proposed a method for finding program faults based on a given property. The idea was to generate an example that negates the property, that is violating it, hence proving the existence of a fault. To facilitate automatic analysis, the specifications are written in a Z-like language, called Alloy, that is strictly first-order. The program statements are encoded as constraints before and after states. A conjunction of the encoded constraints with the negation of the property is then constructed. A constraint solver is then used in an attempt to check the satisfaction of these final constraints. Satisfaction of the constraints is indicative of the violations to the specifications.

Finite-State-Based Testing

Many systems can be described using state machines. That is, operations performed on these systems affect their internal states. These state machines can consequently be very useful when it comes to testing those systems, and how they behave under different conditions. This has attracted various researches in the field. In [153], Chow proposed a method of testing the correctness of a control structure given that the structure can be modeled by a finite-sate machine. The tests were derived from the design and then used to validate whether the specifications are satisfied. In [103] Gaudel presented program testing based on formal specification. The idea was to utilize finite-state machine for testing while relying on some minimal hypotheses on the program under test. In [134], Hierons investigated the cost of using finite-state machines for testing, considering that a known faulty sequence is given. More specifically, their work considered the possibility of having reset transitions in the input sequences. Such transitions would bring the implementation under test to an initial state. They showed that it is sometimes desirable to check sequence with a minimum number of reset transitions rather than a shortest checking sequence.

2.3.3 Security Policy Specifications

In order to test a system against a particular property, the property must be specified. Such specification should be made in a formal way. For example, natural languages are not suitable for such purpose since different interpretations can be given to a single specification. Various proposals were given for the purpose of formally specifying security properties. In [72] Aktug and Naliuka presented an automata-based language, called ConSpec, for policy specification. The language has the advantage of being suitable for the validation of both system's requirements and security-relevant behavior. Another advantage is that it has a reasonably simple semantics. This simplicity however trades off the expressiveness of the language. The language hence allows for the specification of a limited class of policies. In [154], Erlingsson proposed a security policy specification language, referred to as PSlang, as part of a security enforcement toolkit called PoET. The policies of PSlang implement security updates occurring at security-relevant actions, and define any needed remedial actions that should be taken upon validation. The language has the advantage of allowing security policies to be specified in a simple way. Additionally, the language allows additional user-provided actions to be easily incorporated into the target program, hence facilitating monitor inlining. However, this provided simplicity has the drawback of making the specification less formal. The language encodes security automata from a provided text where state variables and updates represent automata states and transitions accordingly. The language does not indicate however how his extraction is performed. This is non-trivial since the provided updates may be given using a programming language that allows for powerful and relatively complicated constructs. In [94, 95], Bauer et al. introduced a language and a system to allow the enforcement of security policies on untrusted Java applications. The language allowed for the composition of complex run-time security policies. Two types of methods, categorized as query/suggest and execute suggestion, comprise these policies. A method of the first type is queried when the application attempts to execute a security-sensitive action. The method returns one of six suggestions indicating how the action should be handled. These suggestions include halting the program or inserting or suppressing some actions. Methods of the second category are then called to execute the updated based on the given suggestions. The work is presented in this proposal is very useful for specifying security policies; however it focuses on Java applications in particular. Additionally, the semantics of the language are not simple, which represents an additional overhead for the security analysts. Furthermore, the correctness of the Polymer policy inlining cannot be proven, as its semantics is not formally presented [72].

2.3.4 Formal Verification and Testing

While formal verification and testing are distinct fields, they can be quite complementary to each other. Formal verification, through its underlying mathematical techniques, can provide a substantial support to software testing. There is a reasonably large spectrum of mathematical techniques including Satisfiability problems (SAT) solving and constraint solving, theorem proving, constraint logic programming, and temporal-logic model checking. In [73], Lynce examined the existing solvers and pointed out that newer solver are capable of solving very large and very hard real-world problem instances. In [37], Detlefs et al. detailed an automatic theorem prover for program checking, namely Simplify, which combines decision procedures for various theories, and employs a matcher to reason about quantifiers. Simplify matches up to equivalence in an E-graph, instead of conventional matching in a directed acyclic graph. This allows it to detect many relevant pattern instances that would be missed by the conventional approaches. In [85], Marriott et al. provided a comprehensive introduction to the subject of constraint programming and, in particular, constraint logic programming. In [48, 49], Clarke et al. provided an important initiation to the field of model checking. Using algorithmic means, model checking can determine whether an abstract model satisfies a formal specification expressed as a temporal logic formula. If this concludes that the specification is not satisfied, a counterexample is provided, which shows the cause of the problem. Furthermore, the fact that the model-checking process can be fully automated, they can be viewed as the most relevant of the above approaches in relation to testing.

Model-Checking and Testing

For the purpose of testing, it is important to model systems' requirements using temporal properties. Properties that allow only a description of current events are insufficient, since they are incapable of providing information on previous system conditions, which may be necessary to detects violations. Model-checking uses decision procedures to find out whether a model satisfies temporal properties. These decision procedures are used to systematically explore the system's space [50]. In [108], Vardi et al. described an automata-theoretic automatic verification of concurrent finite-state programs by model-checking. The point is that if the system model is finite, then exploration can be conducted automatically using model-checking algorithms as has been iterated in [48,49]. The models used by model-checkers can be obtained directly from the source code if available, or alternatively, they can be constructed or built by the user. For the purpose of automation, automatic construction from source code, if possible, is desirable. Examples of model checkers are the Java Symbolic Analysis Laboratory (SAL) [42], that supports dynamic constructs such as object instantiations and thread call stacks then conducts exhaustive verification for the detection of deadlock and assertion failures. Other tools include BLAST [36], an automatic verification tool for checking temporal safety properties, particularly memory safety, in C programs, and SPIN [60], a generic verification system that supports the design and verification of asynchronous process systems. It is worth noting that the process of checking temporal properties can also be applied at run-time, which is then referred to as run-time monitoring. In [22], Artho et al. proposed a framework that provides further automation of test-case generation, based on systematic exploration of the input domain of the program, with run-time verification. Using properties expressed in temporal logic, the execution traces are monitored and verified against such properties.

There are however some important issues that need to be examined in relation to model-checking. One of such is the problem of state-space explosion. Basically, the larger the system grows, in terms of its components, the more chances that the number of states will also grow. The growth may become exponential resulting in the space explosion problem. Another main issue with model-checking is how test-data sequences can be generated. In [59], Fraser et al. argued that model checkers suffer several drawbacks when they are used for testing. This is due to the fact that they were not originally meant to be used for testing and test-data generation, rather for formal verification. They pointed out that significant improvements can be achieved in regards to test suite quality and performance if model checkers are designed with software testing in mind. In [77], Callahan et al. proposed that conformance testing can be achieved by targeting the negation of the property to be tested. Assuming that the system has already been successfully model-checked against a given a property $\mathcal{P}$, instead of attempting to generate input sequences that satisfy $\mathcal{P}$, re-model-check the system again against the negation of $\mathcal{P}$, $\neg\mathcal{P}$. Since the system has already been proven previously to satisfy $\mathcal{P}$, the verification produces input sequences that actually violate $\neg\mathcal{P}$. The reported data are hence the desired test data.

Regardless of the original difference in scope between system verification and testing, a reasonable effort has been made during the past few years to allow the use of model checkers for testing. In [135], Hierons et al. pointed out that formal specifications and testing should not be viewed as rivals, and showed various ways of how formal specifications, particularly model checking, can be used to support testing. Examples of initiatives that combines model-checking and testing are BLAST [36], MOPED [88], a model-checker for linear-time logic (LTL) on pushdown systems, Bandera [75], a tool for reasoning about the correctness of requirements of Java programs, and SLAM [149]. In [150], Ball et al. indicated that SLAM scaled, evolved and matured as the core engine inside Microsoft's Static Driver Verifier (SDV) tool. The latest version of SDV that ships with Windows 7 includes a new version of the SLAM engine. In [47], Gunter et al. illustrated how model-checking, testing and verification can work together for the purpose of unit testing. In specific, they presented a symbolic model-checking approach that can either verify a single unit of code; i.e. a procedure, or a collection of interacting procedures. The proposed approach can be used to automate test case generation for unit testing.

It should be noted that other approaches in relation to model checking and testing have been proposed. For instance, Peled presented in [40] various combinations of model checking and testing techniques so that systems can be checked even if the model is not given, or it is incomplete or inaccurate. In [41], Peled et al. investigated the subject of testing property staisfiability over implementations of unknown structures. They suggested various algorithms to formalize the idea of black-box checking. In [8], Groce et al. investigated the case when both the system and its model are accessible but there are inconsistent with each other. They showed that in such cases, automatic verification can still be possible through the utilization of black-box testing and machine learning.

Security Testing

Software security testing is a particular class of software testing, where the testing focus is not on just finding faults, but rather on finding those particular faults that violates security properties. Such violations mainly compromise systems' security. This contrast the general case of software testing where existing faults may not violate in any way the security aspects of the systems.

In [68], Song et al. pointed out that according to CERT, 10 particular types of vulnerabilities represent the majority of software defects. They referred to these defects as typical defects, and proposed a security testing method, where a threat tree is constructed then traversed based on depth-first search to generate the needed test-data. While this is a reasonable effort, their work mainly targets low-level vulnerabilities such as memory leaks, buffer-overflows and illegal string formatting. In [163], Hui et al. argued that defect-based testing is more effective than specification-based testing. They categorized security errors and investigated existing common error reporting in order to create a security defect taxonomy, which they correlated with most know sever errors. While such taxonomy may be very useful for testers as they argued, the defects included in this taxonomy are mainly safety properties, which are input validation, access validation, concurrency, error-handling support of environment variables, and design defects. In [83], Romero-Mariona et al. pointed out the importance of trustworthiness of software systems and indicated that testing of security-intensive systems is proven to be complicated in many cases. They raised the point that if both safety and security are considered at early stages of the software development cycle then it is more likely that the implemented software will have a reasonable level of trustworthiness. They additionally argued that while there are techniques for security specification, there is a lack of using such requirements in a useful way during testing. They hence proposed a security requirement engineering technique, namely SURE. Using specific syntax, SURE is able to map security requirements into testing artifacts. While the general idea is useful, it has a limited scope since the focus is on early specification and requirements. On the other hand, this lacks focus on already implemented software that is known, or suspected, to be untrustworthy. In [43], Zhang et al. presented a trace-based security testing approach for detecting vulnerabilities in C code. The approach is based on symbolic execution where program traces are executed, symbolically, to produce both program and security constraints. Program constraints are imposed by program logic on the variables, while security constraints are defined as conditions over these variables. Such constraints must hold true for the system to be considered secure. While the approach may indeed be effective in some cases, it has shortcomings including the static nature of the analysis and its lack of multi-language support since it is only usable over C programs. In [131], Hassan et al. showed both the usefulness of formal methods in relation to security testing, as well as the lack of elaborate formal requirements. They indicated that such lack is a major reason behind the difficulty in reaching full automation. They proposed a goal-oriented technique, namely FADES, that bridges the gap between the goal-oriented semi-formal Knowledge Acquisition for autOmated Specifications (KAOS) framework and the B formal method. Among the main issues considered by their proposal was automating the transition from requirements to specifications, as well as automating the derivation of acceptance test cases suite from the requirements. These cases are needed to verify whether the implementation corresponds to the requirements. While the proposed approach can be useful in achieving higher automations, it suffers multiple limitations including an increased initial cost of development, compared to informal approaches, and the lack of requirement completeness and consistency, which may affect the overall design quality. In [66], Haralambiev et al. focused on source code analysis for the purpose of vulnerability detection, particularly over large software. In his research, multiple tools have been used, analyzed, and compared. While this may represent a useful contribution, there is a need for the testers to grasp the business logic of the software being tested. This overhead may not be acceptable in many cases. Further, the proposed approach is heavily based on static analysis, which limits the general capabilities of such an approach. In [27], Yan and Dan proposed an approach to security testing based on what they referred to as a privilege scenario. Their approach is mainly based on model checking, which is actually used to construct these privilege scenarios. It takes these scenarios as input then provides test cases out of them. The privilege scenarios are provided as automata written in SMV language, which is expected by the NuSMV [7] symbolic model checker. Although their approach benefits from model-checking and the ability to formally specify privilege scenarios, it lacks the support of complex scenarios. Extensions are still needed for the approach to be able to specify more complex scenarios as well as to reduce the overall testing cost. In [118], Shahmehri et al. proposed an approach for vulnerability detection based on passive testing. The main idea is to observe the behavior of the system at run-time to conclude the existence of vulnerabilities. The approach examines the execution traces in an attempt to detect the presence of vulnerabilities in these traces. Formal models are used to identify the causes of the detected vulnerabilities and their dependencies. The approach has the obvious advantages inherited from passive testing, that the generation of specific test inputs are not needed. However, there are drawbacks to such an approach including that it is not conclusive. That is, the absence of vulnerability indications in the traces does not mean that the vulnerabilities are nonexistent. Furthermore, there is a difficulty in specifying and testing against a particular vulnerability. There is also an overhead involved since many runs may be needed before the software finally executes a specific vulnerability under concern, hence having an execution trace that proves the existence of such vulnerability. In [67], Shahriar and Zulkernine pointed out the lack of a comparative study of vulnerability detection techniques and suggested a criteria for analyzing software security approaches. A total of 7 issues were proposed as the main criteria behind selecting a particular approach, which included the level of the automation, vulnerability coverage, and test-data generation capabilities. They provided as part of their effort a comparative study where they summarized existing security testing work. While this is a good initiative that could be helpful to security analysts, their work however was based on particular low-level vulnerabilities such as buffer-overflow, string formatting, and SQL injection.

2.4 Software Vulnerabilities Detection Techniques

Understanding the seriousness of security problems, various approaches have been proposed for detecting vulnerabilities in software. While these approaches and their underlying technique may vary greatly, they can be classified as either static-based or dynamic-based. This classification is driven by one major factor, which is whether or not program execution is required for the detection process. In this section, we highlight some of the main proposed approaches for detecting vulnerabilities in software.

2.4.1 Static-Based Vulnerability Detection

Static-based detection approaches attempt to uncover security vulnerabilities by merely investigating the software semantic representation obtained from its structure or documentation, without the execution of the program. The following introduces some of the main static-based approaches proposed for vulnerability detection.

Expression and Statement Pattern Matching

This type of techniques depends on pattern matching of program constructs to uncover vulnerabilities [157]. The pattern is initially stated by the analyst. The pattern matching techniques then scan the software for expressions matching the given pattern. The detection of such expressions reports the existence of security vulnerabilities. Examples of patterns are: `fun1()`, `fun1("str1")`, `fun1(!"str1")`, `x = fun1()`, and `x = y`. The `fun1()` pattern indicates search for any call to a method called `fun1`, which takes no parameters. The `fun1("str1")`, `fun1(!"str1")` calls respectively indicate the search for a method called `fun1` which accepts, or does not accept, an argument value equals to `str1`. The `x = fun1()`, and `x = y` respectively indicate search for assignment expression where the right-hand-side is equal to a value matching the pattern fun1() `fun1()`, or to another variable `y`. This type of pattern matching technique can be useful for detecting vulnerabilities resulting from the use of unsafe calls. For instance, the standard C library provides a large set of calls, such as `strcat()`, `strcpy()`, `gets()`, `scanf()`, `sprintf()`, etc., which are known to be unsafe due to their potential of producing different vulnerabilities, such as buffer overflow. The utilization of such calls can be detected through this pattern matching technique.

It should be noted however that some of the reported vulnerabilities may actually be false-positives. Another problem with this technique is that it focuses on pattern matching rather than on the violation of a security property. Since security properties often involve multiple actions at different points in the program, only few properties can actually be specified through pattern matching techniques. This directly limits the detection power of such techniques.

Verification Through Program Annotation

With this technique, security properties are annotated into the program being tested as sets of pre- and post-conditions constraints. The technique statically propagates through the source code verifying the accumulated set of constraints against the ones defined in the post-conditions. A mismatch would then conclude a violation of the property. One of the main advantages of program annotation techniques over pattern matching is their capability to specify security properties. Additionally, since the annotations are actually described as pre- and post-conditions to program segments, the analysts are able to easily define properties that are program-dependent. On the other hand, these techniques suffer major disadvantages. The annotations are mainly performed manually by the analyst. For real-life software of reasonably large sizes, the needed human involvement becomes intense, resulting in these techniques being very impractical to use. Moreover, due to the static nature of these techniques, false positive reporting is another major disadvantage.

Extending Type Systems

In computer science, a type system is the entity responsible for associating *types* with the values that are the result of expression evaluation. High-level languages use type systems in order to associate types with variables and expressions and to make the program safer. Generally, a type system consists of a set of typing rules and a set of type inference rules. The first set of rules defines types and associates them to language objects and expressions. Type inference rules deduce the type of an un-typed expression in a given program. To prevent runtime errors, type systems are augmented with type constraints that enforce pre-conditions on inference rules. If all type constraints of a program are solved, the program is "*well-typed*" and is free of runtime errors. This feature is summarized by the famous 1978 Milner's slogan "*well-typed programs cannot go wrong*" [113].

Different programming languages handle type errors differently. In general, type safety defines the extent that a programming language discourages or prevents type errors. As a matter of fact, type safety can sometimes be controlled by the program, rather than by the language of that program. In other words, the type safety provided by the languages can be overpowered by careless programming.

Type errors, such as improper explicit casting, can be the source of many security vulnerabilities. This group of verification techniques targets such violations. Some sort of type enforcement is hence needed. This enforcement can be conducted statically through extending type systems to the compiler, hence detecting potential vulnerabilities at compile time. While these techniques can be very effective in detecting this type of vulnerabilities, they have major drawbacks. The scope of vulnerabilities that can be detected by such system is very limited [65]. Additionally, since the security properties are actually defined in the compiler, the techniques are incapable of allowing further user-defined properties.

Model-Checking

One of the main problems with the aforementioned approaches for static vulnerability detection is their lack of automation. Pattern matching is only capable of describing a very limited set of properties. With program annotation, a huge user involvement may be required to annotate the pre- and post-conditions. The implementation of type system and enforcement in the compiler is by no means a trivial task.

Model-checking [107] belongs to the field of logic in computer science which attempts to solve the following problem: Given a model of a system, automatically test this model to discover whether or not it complies with a given specification. Consequently, model-checking can be very suitable for detecting security violations. Techniques based on model-checking allow the specification of security properties, often through finite-state machine representation. The techniques scan the software being tested and matches changes in its states to the given property. A vulnerability is flagged if the trace results in the property specification being violated (i.e. reaching an error or unexpected state of the state machine).

We view model-checking as one of the most useful techniques for statically detecting security vulnerabilities in software. In addition to the automation capability, there are other advantages to model-checking techniques. Model checkers function over an abstracted model of program states, so they are independent of the programming language of the program being tested. An existing model checker can be made capable of detecting vulnerabilities in programs written in newer languages by only implementing a parser for the new language. However, being based on static analysis, model-checking techniques suffer the major problem of false-positive reporting.

2.4.2 Dynamic-Based Vulnerability Detection

Due to the nature of static analysis, detection techniques based on such analysis are often inconclusive and suffer major problems including the potential of reporting a massive number of false-positives. Dynamic analysis approaches address the limitations of static analysis. In contrast to static-analysis approaches, dynamic analysis techniques require program execution for detections to take place. Various dynamic analysis approaches exist. Below, we highlight some of the major ones of such approaches.

Random-based Verification

The main idea behind this approach is to execute the program repeatedly with randomly generated data until either the vulnerability is detected or a pre-determined number of executions is exhausted. For instance, if detection is to take place against buffer overflow vulnerabilities, then the approach would randomly generate long strings for program input. The execution of the program would take place over these strings in an attempt to detect the vulnerabilities.

Using this approach may very well reveal security vulnerabilities in programs. The approach clearly has a major advantage, which is its simplicity. It is very easy to implement and apply. However, the approach suffers many drawbacks. Firstly, the set of security properties that this approach can test is quite limited. Secondly, the approach is not sound. The lack of positive reporting does not guarantee the non-existence of vulnerabilities. Additionally, the reporting of errors does not guarantee that all vulnerabilities have been discovered. Thirdly, the time spent to repeatedly execute the program before detections are made can be extensive. Due to the random nature of the approach, many executions may take place over traces/paths of the program that do not include the vulnerabilities. Consequently, detection tools based merely on this approach are considered unreliable and inconclusive.

Directed Random-Based Verification

This approach attempts to mitigate some of the problems of the random-based approach. Instead of generating data that may take the program execution towards any path, the approach attempts to generate data that would drive program execution towards specific paths. The choice of these paths is critical and usually based on the sensitivity/importance of these paths, or on the potential existence of vulnerabilities in these particular paths. For instance, if 20% of all programs paths are possibly those ones where 90% of the targeted vulnerabilities may exist, then it is wise to keep in generating data that would execute these paths, at least first. This contrasts the random-based approach, where possibly most of the executions may actually end up taking place over the undesired paths. While this approach may provide some improvements over the mere random-based approaches, it still suffers the same main problems due to its random nature.

Fault Injection Verification

Fault injection techniques attempt to enhance the testing coverage of the program by actually injecting faulty code segments at particular program points. Those points are usually selected very carefully to provide the needed detections. Such points may include error handling paths that may rarely execute. The approach mainly considers that vulnerabilities are caused by program faults, which drive the execution through error handling routines or assertion failures. The injection process can be automated by locating those program points where faults would drive execution towards fatal error segments including assertion failures. Faulty code is then injected at these locations, and program execution is attempted. Failures in such execution would be indicative of security vulnerabilities in the program.

While this approach may uncover some vulnerabilities, it has a major drawback caused by its fundamental idea of injecting faults that the program does not actually have. In other words, it is very questionable whether the detected vulnerabilities are indeed exploitable.

Detection through Dynamic Monitoring

The aforementioned approaches detect vulnerabilities by executing the program and mainly leading it to a crash of some sort. While this may indeed flag the existence of a vulnerability, it definitely does not show the violation of a particular security property. As a matter of fact, none of them allows precise specification of the property.

Vulnerability detection through dynamic monitoring differs from all these approaches. The main idea of such an approach is to allow the analysts to specify the security property that they need to test the software against. Runtime monitoring routines are then instrumented at particular program points that relate to the specific property being tested. When program execution takes places, the monitors verify whether the execution is in compliance with the property. If not, then this is immediately flagged as security violation to the property being tested. We generally view this technique as the most prominent for dynamically detecting security vulnerabilities.

2.5 Code Instrumentation

In order to obtain profiling information of a program, it is necessary to inject monitoring code at specific locations, with which behavioral information can be collected. While the general idea of code instrumentation is rather simple; that is just inject additional monitoring code in the base code, various code instrumentation techniques exist. These techniques significantly differ from each others depending on where the monitoring code is to be injected and the purpose behind such injection. Below, we provide a background of the major existing approaches for code instrumentation.

2.5.1 Preprocessor's Macros Instrumentation

Many programs, such as compilers, operate on data that has been previously processed by other programs, the preprocessors. The lowest-level of such programs are referred to as lexical preprocessors, as they only require lexical analysis. Lexical preprocessors typically perform macro substitution, textual inclusion of other files, and conditional compilation or inclusion. The most widely used lexical preprocessor is C PreProcessor (CPP) used by C and C++ compilers. When programs are written in these languages, preprocessor directives are added to the code; these are lines that are not program statements, rather directives for the preprocessor. The preprocessor handles directives for source file inclusion (#include), macro definitions (#define), and conditional inclusion (#if). Applying C preprocessor macros for code instrumentation has been proposed by Kranzlmueller [93]. The basic idea is to replace the original statements to be observed by monitoring code that would effectively perform the same activities of the original code and additionally provide the needed monitoring functionality. In practice, this can be done by providing a macro definition for each event of interest, which maps the original function onto a monitor function.

2.5.2 Library Inclusion Instrumentation

Code instrumentation can simply be achieved through implementing a library that incorporates all the needed monitoring information. Calls to this library from an original code would hence imply code instrumentation. The advantages of such technique lie in its simplicity, its minimal requirements to alter the original code, as well as its portability and hardware independence. However the technique requires re-linking of the application.

An example of such a technique is the Message-passing Portable Instrumented Communication Library (MPICL) [127], a subroutine library for collecting information on communication and user-defined events in message-passing parallel programs written in C or FORTRAN.

2.5.3 Library Replacement Instrumentation

This technique is similar to library inclusion, however instead of adding a new library, the techniques modifies the original library used by the program to include all the needed monitoring information. A re-linking is then needed to use the modified library. The technique is still simple; however it does not allow monitoring of program behaviors except for those associated with the external libraries. Another disadvantage of this technique is that it also requires that the original functionality must be coded in the replacement library. This large overhead questions the applicability of this technique in reality.

2.5.4 Parser-Level Instrumentation

A parser is the component of a compiler that carries out the task of analyzing a sequence of tokens to determine its grammatical structure with respect to a given formal grammar. The parsing process also transforms the data into data structure, usually an abstract syntax tree (AST), which captures the implied hierarchy of the input. A parser hence has access to additional information that is unavailable to preprocessors. Consequently, utilizing parsers for code instrumentation is more advantageous. The AspectC++ Puma library is one example of such parsers, which is capable of performing a large spectrum of complicated code transformation at various points of a program. However, since parsers naturally transform source code to data structures, i.e. AST, the technique is disadvantageous when original-source-to-instrumented-source code transformation is required. Although this is still possible, it requires additional transformation steps, which lead to increased compilation time; an overhead that many applications may not accommodate.

2.5.5 Binary Wrapping of Object Code

As discussed, preprocessor's macro instrumentation does wrapping of textual information of a program into other textual representation. This technique can be viewed as an enhanced version of such textual substitution techniques. Instead of wrapping textual representation, the technique operates by changing the executable code of a program by substituting the bytes of the object code by others. Examples of tools utilizing such techniques are the Just in Time Instrumenter JiTI [136], Performance Analysis Tool (PAT) [61] and Analysis Tools with Outcome Mapping (ATOM) [148], which could insert calls to arbitrary C functions before/after individual instructions, basic blocks, or functions of the program being instrumented. A slight variation of this technique exists, which performs the wrapping on the executable code instead of the object code. The Executable Editing Library (EEL) [97] is an example of applications that utilize this instrumentation technique.

Compiler-Assisted Instrumentation

This technique enhances the parser-level code instrumentation technique. The difference between the two techniques lies mainly in the processing stages. With this technique, once the abstract syntax tree, or parse tree, is generated, no generation of transformed source code is performed; instead, the tree is directly passed to the compiler's backend and object code of the instrumented program is directly generated by the compiler.

Despite of the overhead needed for transforming the intermediate representation, the direct processing of these representation makes this technique both powerful and efficient. Protagoras [29], a plug-in architecture for the GNU compiler collection that allows one to modify GCC's internal representation of the program under compilation, is an example where this technique has been applied.

2.6 Test-Data Generation

In order to prove that specific vulnerabilities exist in the software being tested, the software must be executed with concrete data values, with which the program is driven towards the execution of the vulnerabilities. Traditionally, the creation of this data set was performed manually. However, with the obvious increase in complexity and size of software during the past decade, manual creation of this data turned out to be a next-to-impossible task. Thus, it was clear that automation is significantly needed. This has attracted many researchers in the field. Consequently, various, and significantly different, approaches to test-data generation have been proposed during the past few years. The capabilities of these approaches indeed differ. In fact, in [81], Miller et al. pointed out the difficulty being faced in accepting these techniques in a general form. In this section, we provide a background of some of the most prominent approaches to test-data generation. We firstly provide a literature review of the subject and its related approaches and proposals. We then follow with a more detailed description about some of these approaches in particular.

2.6.1 Automatic Test-Data Generation

The subject of automation represents a crucial aspect when it comes to test-data generation. There are many reasons behind that including cost effectiveness, and the avoidance of the need of manual intervention. In [78], Edvardsson argued that Automatic Test-Data Generation (ATDG) is among the most important components of software testing. While attempts can be made to classify the proposed techniques, it should be noted that there is no definite isolation between the underlying methodologies of these approaches and that overlaps of techniques are sometimes present. For instance approaches based on path analysis may very well use constraint solving to achieve what is needed.

Path-Based Techniques

This type of approaches focuses on path analysis for test-data generation. In [130], Dharam and Shiva pointed out that in spite of the fact that various software testing techniques exist, they do not insure that all possible behaviors of the software are analyzed, executed, and tested. They proposed dynamic-monitoring based approach for the purpose of detecting and preventing path traversal attacks. The approach combined two pre-deployment testing techniques: data-flow testing and basic path testing. While the proposed approach may have the potential of being useful for path-based testing, further work is still needed in terms of both automating the proposed process, as well as extending it for the detection of a more complex attacks. In [119], Sy and Deville proposed a technique for ATDG for imperative programs containing integer, boolean and/or float variables. The goal of the generated data is to execute a specific statement, to traverse a branch or to traverse a specified path. For path traversal, the approach transforms the path into constraints then solves them using an interval constraint-solving algorithm. The desired test-data is then obtained from the interval solution. For statement or branch coverage, the approach dynamically constructs a path that reaches the specified statement or branch then uses the path traversal technique to generate the required data. In [76], Lin et al. proposed a technique that extended the Hamming distance to measure the distance between two paths. The approach also utilizes genetic algorithms by employing a fitness function to achieve the goal of the selected path. In [126], McMinn et al. introduced a search-based approach to ATDG, referred to as species per path. It transforms the program to a version in which multiple paths to the search target are factored out. A group of operating species is then run in parallel to generate the data. The point is that the factorization of the paths results in different search landscapes. These landscapes are potentially more conducive to test data discovery than the original overall landscape.

Heuristic Search Techniques

These techniques are based on based on heuristic search methods or evolutionary computation, such as simulated annealing and Genetic algorithms. In [117], Mansour and Salame, proposed two stochastic search algorithms for generating test cases for specific path execution. One of the algorithms was based on simulated annealing, while the second was based on genetic algorithms. Both algorithms however were based on optimization formulation of the path testing problem. The targeted paths included both integer and real-value test cases. In [120], Tracey et al. proposed an approach that targets the automation of full-path coverage of a given structure. Their approach resorted to heuristic-global optimization and simulated annealing. Their data generation tool considered boundary value analysis, where test-data that lies close to the boundaries of a given subdomain is to be considered of a higher adequacy than test-data further away. Their tool also considered assertion/run-time testing, where the generated data is to break program assertions or cause runtime exceptions, and component re-use testing. Component re-use testing aims at finding test-data that causes the precondition of a re-usable component to be broken, possibly as a result in changes of environment conditions between use and re-use times. In [125], Bueno and Jino presented a tool for test-data generation and identification of structural testing. The proposed tool used genetic algorithms and employed a fitness function that combined dynamic control and data-flow information to speed up data generation. In [161], Liu et al. proposed test-data generation approach that takes advantage of a flag cost function. This cost function is introduced as the main component of fitness function, whose value changes with the variation of the flag problem. The proposed approach took advantage of the multi-agent genetic algorithm to find, with a small cost, test-data for program statements along with the flag problems.

Graph-Based Techniques

Various proposals suggested the construction of graph that then guides the test-data generation process. In [132], Pargas et al. suggested an algorithm based on genetic algorithms. They implemented a tool for parallel processing to improve the search performance. The direction of the search was guided by the control-dependence graph of the program. In [96], Gallagher et al. focused on object-oriented testing. They extended single class testing to multiple-class integration testing. A single class behavior is modeled as a finite state machine. The approach was to transform the representation into a data flow graph that explicitly identifies the definitions and uses of each state variable of the class. They extended this idea to inter-class testing through the construction of flow graphs, finding paths between pairs of definitions and uses, detecting some infeasible paths and then automatically generating test-data for these glasses. In [26], Yang et al. focused on automatic and semi-automatic testing for parallel programs. They demonstrated that with some extension, sequential test-data adequacy criteria can still be applicable to parallel program testing. To support their approach, they utilized parallel program flow graph. In [139], Gouraud et al. proposed an approach for automating statistical structural testing, based on the combination of uniform generation of combinatorial structures and randomized constraint solving techniques. The idea was to formalize the control flow graph of the program as combinatorial structure specification. This allowed for execution paths to be drawn uniformly. The drawn paths are then analyzed and test-data are obtained using a constraint solving library. In Gupta et al. [116] and Soffa et al. [106], the authors presented an approach that targets test-data generation for a given path. The approach attempted to iteratively refine the input using a set of linear constraints on the input. They designed an approach, namely the Unified Numerical Approach (UNA), to solve the constraints. UNA constructed and utilized control flow graphs of the program to generate test-data. In [140], Edwards outlined a general strategy for automated black-box testing. This is achieved using directed flow graphs, where one end of the graph represented object construction and the other represented object destruction. Hence, the graphs simulated the lifetime of the object and the allowed sequences of operations over these objects. In [5], Sofokleous and Andreou proposed a dynamic framework for test-data generation. The framework was based on genetic algorithms. The idea was to have one component for analyzing the code being tested. This component then interacts with a test-data generator to automatically produce test-data. The analyzer's function included the extraction of the related variables and statements from the program and the creation of control-flow graphs accordingly. The Generator component attempted to produce a near-optimum set of test-data cases with respect to edge/condition coverage criterion. For that, the generator took advantage of two optimization algorithms, the Batch-Optimistic (BO) and the Close-Up (CU). A comparison to other techniques, in terms of time and coverage, was given. While the proposed approach represents a reasonable contribution to the subject, it suffers some limitations. The analysis is based on control-flow graphs of individual units of the program. That is, a separate control-flow graph is constructed for each function/method of the program being tested. A new control-flow graph and a new required set of test cases are initiated for each method. This overhead is directly increased as the number of methods in the program increases. Another limitation is in relation to testing object-oriented programs. Since the control-flow graphs are constructed for individual methods, these control-flow graphs are unable to describe, and hence test, OO class-based features.

Syntax-Based Techniques

This type of testing takes advantage of information that can be obtained by investigating the program's syntax. This includes for instance functional analysis, boundary value analysis and partitioning analysis. In [10], Hajnal and Forgács proposed an integrated testing criterion that extended traditional criteria in order to effectively reveal domain errors. An automated test-data generation algorithm is developed to satisfy the criterion. The proposed algorithm combined path selection and test-data generation, where function minimization is used to find the required test cases. The algorithm first initiated coverage of some predefined program paths and then a domain testing is conducted for additional requirements of some ON-OFF points on these paths. In [104], Gallagher and Narasimhan presented a software system for generating test-data for programs developed in Ada83, namely ADTEST. ADTEST located input domain boundaries intersections and generated ON-OFF test-data points. Boundary value analysis was also used as one of the tests described in [120].

Numeric-Based Techniques

These techniques take advantage of on numerical or mathematical approaches in order to provide the needed test-data. For example, the key feature of ADTEST, presented by Gallagher and Narasimhan in [104], is that the test-data generation problem is treated entirely as a numerical optimization problem. This avoids shortcomings presented in other techniques, such as symbolic executions, including input variable-dependent loops, array references, and module calls. In this case, program instrumentation was used to solve a set of path constraints without requiring an explicit knowledge of the exact form of these paths. The approach presented in [96], also takes advantage of such techniques. The approach allows database tables to be associated with a mathematical set, that is defined as the sequences comprising the primary key values of the tables.

2.6.2 Random Test-Case Generation

Random Test-Case Generation (RTCG) is one of the earliest techniques that are used for such purpose. The level of randomness in various RTCG techniques vary. For example, some of these techniques may view the entire software to be tested as a single unit. In other words, they do not distinguish between different subroutines or different portions of the code when the tests are generated. They hence create a set of test cases totally randomly and initiate testing based on these cases. Other RTCG techniques would still create the tests randomly but based on assumptions concerning fault distributions, or concerns over specific portions of the code that could be more significant or more susceptible to errors.

Although random tests are relatively easy to generate, they are quite extensive, since millions of tests may be needed before a particular error in the software can be detected. Additionally, they are not sufficient. Even if a mass number of test cases is generated, there is no guarantee that these cases will detect a specific error, or guarantee full-path coverage. Consequently, the applicability of these kinds of test generation techniques is often limited due to this fact and other various factors, such as the size and complexity of the software.

2.6.3 Structural Testing

Sometimes, the code of the software to be tested might be available. In such cases, different types of testing can be performed. This is often referred to as white-box testing or structural testing. Generally, since the code is available for testing, various tests can be performed either statically or dynamically. Below we describe some of the main techniques of structural testing.

Goal-Oriented Approach: It is somehow a contradictory approach to the classical full-path coverage approach. With full-path coverage, each path in the program must be traversed at least once. On the other hand, the goal-oriented approach does not have such constraint; rather it qualifies the various flow control branches of the program with an important level, such as *Critical*, *Semi-Critical*, *Non-Essential*, or *Required* with respect to a specific target. The target is usually a specific node in the Control Flow Graph (CFG), which reflects a specific objective. A simple definition of the approach has been given by Korel [91] as follows: *Given node Y in a program. The goal is to find a program input x on which node Y will be executed*. By traversing the different paths of the CFG, a decision can be made on the appropriate classification of the path. In specific, any path that would lead the execution away from the target node is considered critical and should not be attempted.

The approach starts by generating arbitrary program inputs. A search procedure then examines whether the execution with a value of them at a specific node would lead to the target. If so, the execution continues; otherwise, a new program input must be considered at that node in an attempt to force the execution through the desired path. To illustrate the approach, consider the C++ code fragment given in Figure 1.

The control-flow graph representing the code in Figure 1 is shown in Figure 2. In contrast to full-path coverage approach, go-oriented focuses on essential branches; which are those branches influencing the execution of the target node. The scope of control influence for conditional and looping statements is defined as follows [91]:

a) **if** Z **then** B1 **else** B2

X is in the scope of control influence of Z iff X appears in B1 or B2.

b) **while** Z **do** B

X is in the scope of control influence of Z iff X appears in B.

Classification of branches can then be concluded as follows:

1. *Non-Essential*: A branch (n_i, n_j) is non-essential with respect to a node Y iff Y is not in the scope of the influence of n_i. For instance, assuming that the loop at node 3 in Figure 1 will eventually terminate, i.e., no infinite loop, then branch $(3, 4)$ has no influence

```
void GoOrientedApproach() {
    int i1 = 0, i2 = 0, i3 = 0, i4 = 0, i5 = 0, i6 = 0;
    cout << "Enter 4 Integers: ";
    cin >> i1 >> i2 >> i3 >> i4 >> i5;
    while (i1 < 10) { // node 3
        if (i5==12) { // node 4
            cout << "Point1" << endl;
            i6++; } // node 7
        else {
            cout << "Point2" << endl;
            i1 *= 10; }
        i1++; }
    i2++; // node 10
    while (i2 < 10) { // node 11
        i3 = i2 - 1;
        if (i3 % 2 == 0) { // node 13
            cout << "Point3" << endl;
            i2++;
            while(i3 > 5) { // node 17
                if (i6 > 0) { // node 18
                    cout << "Target Point" << endl; }
                else {
                    i3 -=5;
                    cout << "Point4" << endl; }
                i3–; } }// node 22
        else {
                cout << "Point5" << endl; // node 15
                i2 *= 2;
                i5 += 10;
                 if(i2 > 2) { // node 25
                        cout << "Point6" << endl; }
                if(i3 < 12) { // node 27
                        cout << "Point7" << endl; }
                else {
                        cout << "Point8" << endl; }
                i3 +=2; } // node 30
         i4–;
         i5++; // node 32
    }
    i4 = i3–;
    i5 = i2 + 4;
    if (i4 == i5) { // node 35
         cout << "Point9" << endl; }
    else {
        cout << "Point10" << endl; }
    cout << "Point11" << endl; // node 38
}
```

Figure 1: Goal-Oriented Approach Example

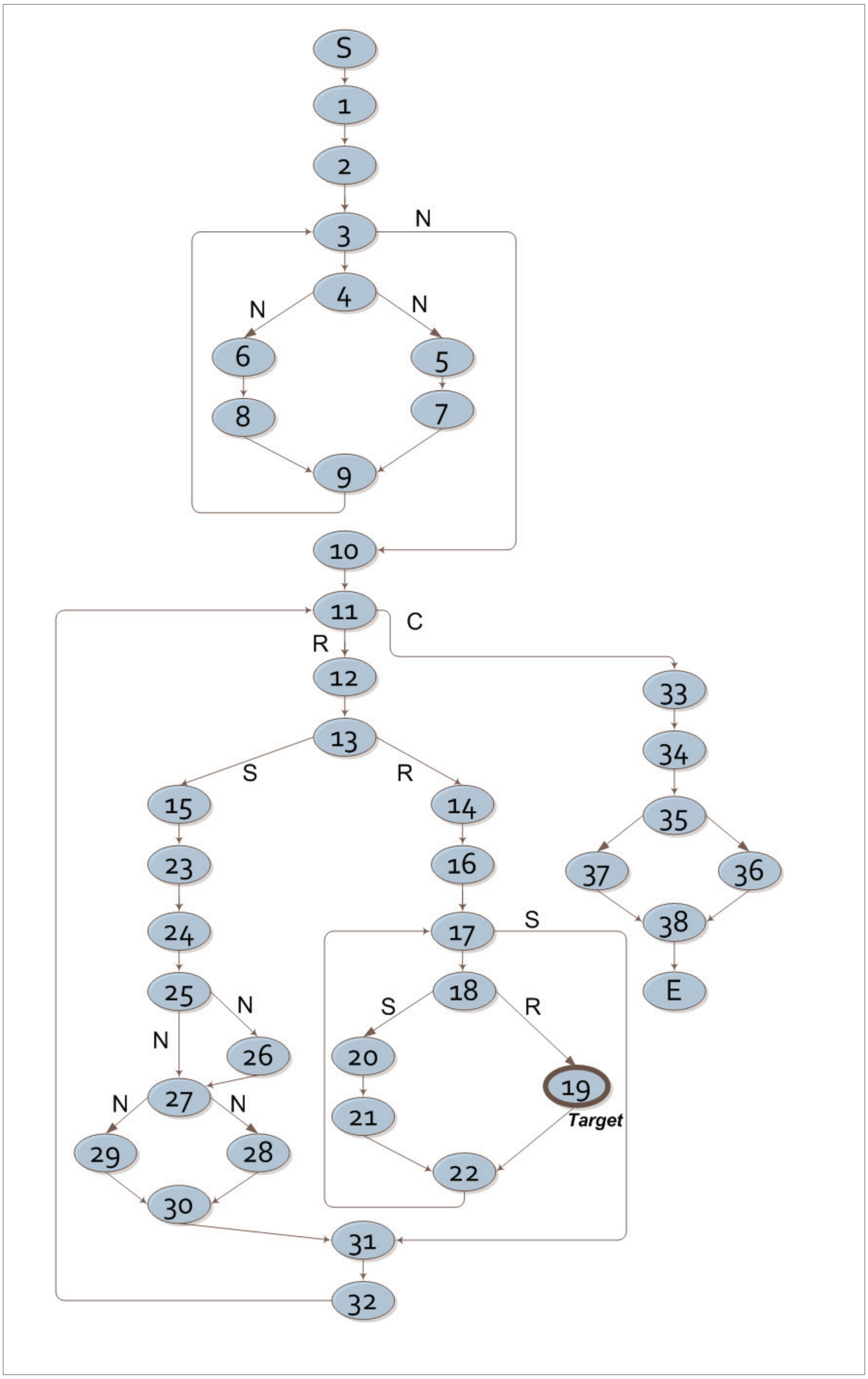


Figure 2: Flow Diagram for Code Segment in Figure 1

on the execution of the target node (node 19). Consequently, this branch is considered as non-essential in relation to the target node. Similarly, branches $(3, 10)$, $(25, 26)$, $(25, 27)$, $(27, 28)$, and $(27, 29)$, are all non-essential. Since non-essential branches do not influence the execution of a target node, execution of these branches should always be allowed.

2. *Critical*: A Critical branch is a branch that would permanently lead the execution away from the target node. A branch (n_i, n_j) is critical with respect to node Y iff (1) Y is in the scope of the control influence of n_i, and (2) There is no possible path from n_i to Y through branch (n_i, n_j). In other words, the execution of branch (n_i, n_j) in such case is critical in the sense that it will immediately and permanently eliminate all chances to reach the target node. For instance, branch $(11, 33)$ in Figure 2 is critical since the execution of this branch would permanently lead the execution away from the target node (node 19).

3. *Semi-Critical*: A branch (n_i, n_j) is semi-critical with respect to node Y iff (1) Y is in the scope of the control influence of n_i, and (2) There is no acyclic path from n_i to Y through branch (n_i, n_j). In other words, the execution of semi-critical branch would lead the execution away from the target, but not necessarily permanently since it may still lead to the target node through a backward edge of a loop. Branches $(13, 15)$, $(18, 20)$, and $(17, 31)$ in Figure 2 are hence semi-critical. Theoretically, since the execution of non-essential branch leads the execution away from the target node, a program executing non-essential branch should be terminated. However, this rule can be "relaxed" since it is still possible to reach the target node even if a non-essential branch is executed.

4. *Required*: A branch (n_i, n_j) is required with respect to node Y iff (1) Y is in the scope of the control influence of n_i, and (2) There is an acyclic path from n_i to Y through branch (n_i, n_j). In other words, required branches are those which must be executed for the

target to be reached; consequently program execution must be allowed to continue through those branches. Branches $(11, 12)$, $(13, 14)$, $(17, 18)$ and $(18, 19)$ in Figure 2 are clearly required for the execution of target node (node 19) to take place.

The Chaining Approach: The goal-oriented approach suffers a major limitation due to the fact that only control-flow graph is used to guide the search process. This limitation may result in some nodes being very difficult to reach, or a massive number of attempts may be needed first before finally finding the proper data to reach them. To illustrate the problem, let us reexamine Figure 2. Although both branches of node 4 are considered non-essential by the goal-oriented approach in relation to the target at node 19, it is actually clear that unless node 7 is executed there will be no way for the target node to be reached.

The chaining approach is an extension to the goal-oriented approach that is aimed to minimize this kind of problems. Before describing the exact details of the approach, it may be helpful to first point out some of the basic concepts as described by Korel *et al.* [54]. Generally, a control-flow graph can be represented as a directed graph $C = (N, A, st, ed)$, where N is a set of node, A is a set of edges (hence a binary relation defined over N elements), st and ed are respectively the two unique entry and exit nodes of the graph. Nodes can further be recognized as either *simple* or *test nodes*. Test nodes are those nodes where conditions must be evaluated, and accordingly different transfers of control take place. Outgoing edges from test nodes are then referred to as *branches*, where these branches are controlled by *branch predicates*, describing the conditions under which a branch is to be taken. An input variable x is a variable that appears in at least one input statement in a program, such as $cin >> x;$. Assuming $I = (x_1, x_2, ..., x_n)$ is a vector of input variables of a program, and D_{x_i} is the domain of input values x_i; that is the set of all possible values that x_i can hold, the domain D of a program can then be defined as $D = D_{x_1} \times D_{x_2} \times ... \times D_{x_n}$. An input *x* can then be viewed as a single point of input, $x \in D$, in the n-dimensional input space D. A path P from node n_{k_1} to n_{k_q} is a sequence $P = \langle n_{k_1}, n_{k_2}, ..., n_{k_q} \rangle$ of nodes such that for all *i*, $1 \leq i < q$, $(n_{k_i}, n_{k_{i+1}}) \in A$. A path is *feasible* if at least one input exists with which this path can be traversed; otherwise the path is *infeasible*. A *use* of a variable is a node where the value of the variable is used. A *definition* of a variable is a node where the variable is assigned a value. Assuming that $U(n)$ is the set of variables that are used at node n, and $D(n)$ is the set of variables that are defined at node n, a definition-clear path can then be defined. A definition-clear path from n_{k_1} to n_{k_q} with respect to variable v is a path $\langle n_{k_1}, n_{k_2}, ..., n_{k_q} \rangle$ that starts at node n_{k_1} and ends at node n_{k_q} and for all i, $1 < i < q$, $v \notin D(n_{k_i})$ $-$ that is *v* is not defined along the path. Similarly, a definition-clear path from n_{k_1} to n_{k_q} with respect to a set of variables, S, is a path where for all i, $1 < i < q$, $(D(n_{k_i}) \bigcap S) = \varnothing$ $-$ that is none of the variables in S is defined along that path.

The basic idea of the chaining approach is to identify a sequence of nodes that needs to be executed before the execution of a specific node. That is, instead of solely depending on a control-flow graph, the approach uses data dependency analysis to guide the test data generation process [54]. The approach initially starts by executing the program with random input. Similar to goal-oriented, the approach decides whether the execution should continue through a branch depending on whether, or not, the branch would lead to the target. If the path does not lead to the target, then a new input value is to be found and attempted at the node controlling the branch. If the search however cannot find such input value, then it attempts to alter the flow at that node by identifying nodes that have to be executed prior to the execution of this node; this contrasts the goal-oriented approach, which would have declared failure by that time. The idea is that by altering inputs at the previous nodes, there is a chance to derive a desired input value at the concerned node.

2.7 Conclusion

In this chapter, we have distinguished between the different fields of security in relation to software and computing. This distinction is made for the purpose of highlighting the focus of the work presented in this thesis, which targets software security. We have introduced a classification of security properties and detailed both low-level and high-level ones. Additionally, we have provided background on two major subjects that are related to this research effort, which are code instrumentation and test-data generation. The different approaches to software vulnerability detection have been introduced. These approaches have particularly been classified as either static-based or dynamic-based approaches. Various static-based approaches exist. However, all of these approaches suffer common limitations caused by the nature of static analysis, such as the potential reporting of false-positives. Similarly, various dynamic-based analysis approaches exist. While the capabilities, effectiveness, and detection powers of these approaches greatly differ from one to another, all of these approaches share common drawbacks, such as the potential of an intense number of executions before detections can be made. Furthermore, the issue of test-data generation becomes very crucial to such approaches, since without being able to generate the appropriate data, execution cannot be driven as desired. In conclusion, it is clear that both static-based and dynamic-based approaches carry their advantages and drawbacks.

We hence need approaches that are rather based on a synergy between the two analyses. Such approaches have the potential of becoming very advantageous to security testing and vulnerability detection. This conclusion represents one of the major foundations of our research, as we explain in the following chapters of this thesis.

Chapter 3

Security Model

3.1 Introduction

The research reported in this chapter builds on the ideas that were initially published in [70, 162]. Actually, the idea of extending the expressiveness of edit automata with team components and combining them is still the same. However, it is important to mention that the semantics ascribed is completely different. In fact, in this thesis, we provide a more rigorous semantics in the sense that is provides a complete formalization of some the ideas that we formulated in the aforementioned publication. Besides, we brought major modifications to the definitions of: configurations, sequents, transitional relations, and therefore semantics rules. As such, the research reported in this chapter deviates, in major way, from the contributions published in [70, 162].

In order for security analysts to test software, they must first be able to specify the security property they wish to test the software against. Hence, a language is needed to allow such specification. Such a language should be expressive, in the sense that it should allow analysts to state the desirable safety and security properties that they wish to. In addition, this language should be formal, i.e., it should be endowed with rigorous syntactic and semantic definitions. In this chapter, we detail our contribution in relation with this matter. Recognizing the significant need for such a language and the lack of an existing one that fully addresses our requirements, we design and implement the Security Concern Specification Language (SCSL). SCSL is designed with many requirements in mind including:

- It should allow the user to specify a large spectrum of both safety (low-level security) and security properties.
- It should allow the user to easily specify the desired security properties through a simple easy-to-learn graphical notation.
- It should be endowed with formal syntactic and semantic definitions.
- It should cater for the composition of several specifications of security properties.
- It should allow for the specification of mitigation measures upon the detection of a given security vulnerability.

However, specifying security properties is not a matter that is only applicable to security testing or evaluation. Other security fields require such specification as well. Various research proposals have previously addressed this issue. In particular, we find that a similar concern has been looked into in the field of security enforcement. A very popular technique in that field is to monitor program execution and then take remedial actions if the execution violates a given security policy. It should be noted that security policy differs from security property. The distinction between property and policy is given in Section 3.3.1. Different mathematical models, such as security automata [142] and edit automata [101], have been introduced by researches. The difference between such proposed models lies mainly in the way they react to violations; for instance some of them would halt the program, while others would suppress the violation, insert further actions and continue execution.

We particulary view edit automata as a very attractive model in relation to our research. The model is proven to be capable of enforcing all safety properties as well as many security properties [101]; hence, it represents a great initial framework for SCSL. However, the model as proposed is not able to address all the aforementioned requirements. Firstly, the model is designed to enforce a single security property. Consequently, the model does not explicitly define an enforcement action if multiple monitors suggest different remedial actions. For the purpose of vulnerability detection, we require a deterministic behavior to be defined based on the group of security properties to be tested. Secondly, since different components in contemporary software may be tightly related, interaction between different monitors is also needed. For instance, disabling and enabling system interrupts by one program unit may very well impact the behavior, and controls the allowable operations by other units. Deleting a dynamically allocated memory space in languages such as C/C++ would invalidate all pointers referencing it. Such examples illustrate the need of interaction between various monitors for performing appropriate behavior. Since an edit automaton is rather an individual unit, it is not possible to achieve this type of interaction capability using edit automata.

3.2 Security Properties and Automata

Finite state machines are quite powerful in modeling both the execution behavior of programs as well as the interaction between different components under execution. Various research proposals have consequently taken advantage of such capabilities.

Security Automata (SA) [142] have been introduced by Schneider. One of the main goals of SA is to guarantee that *nothing bad will ever happen*. This is achieved by monitoring the execution and halting the program should any violation occurs. SA also defines a specific set of properties that can be enforced by such automata, namely safety properties. Edit Automata [101] have been proposed as an extension that enhances security automata such that security properties can be enforced as well.

Other research proposals have focused on particular aspects of SA. Bounded history automata [151] have been introduced as a new class of automata, where the focus is on the characterization of security policies that are enforceable by execution monitors constrained by memory limitations. Other proposals [90] have investigated the computability constraints of the properties being enforced.

Input/Output (I/O) Automata [102] have been introduced as a labeled transition system model for components in asynchronous concurrent systems, with actions defined as input, output and internal. An I/O automaton has *tasks*, and in a fair execution of such automaton, all tasks are required to get turns infinitely often.

The idea of I/O automata inspired the proposal of Team Automata (TA) [53], which actually extend I/O automata. TA form a framework of distributed and reactive components, referred to as component automata. The communication between these components is then modeled by the team. Component-interaction automata [21] have then been introduced based on TA, for the purpose of modeling significant interactions between the components in a hierarchical component-based systems, as well as verifying the system behavior. Timed component-interaction automata [84] have then been introduced. These automata particularly target the verification of components' interaction behavior under timing constraint.

3.3 Team Edit Automata

The primary contribution of this chapter is the proposal, the design and the implementation of a specification language, with which security analysts are able to formally specify security properties that they wish to test the software against. In the following, we first provide the background underlying security automata, edit automata and team automata. Afterwards, we detail our Team Edit Automata (TEA) model, which composes the core of our security specification language.

3.3.1 Security Automata

Alpern and Schneider [12] have distinguished between properties and more general policies as follows. Assume A^* denotes the set of all finite-length sequences of actions on a system with action set *A*, $\Sigma \subseteq A^*$ is a set of executions, ε denotes a single execution, and σ denotes a sequence of actions within an execution. A security policy is a predicate *P* on sets of executions. A set of executions Σ satisfies a policy *P* if and only if *P*(Σ). A security policy *P* is deemed to be a computable property when it is a predicate over sets $\Sigma \subseteq A^*$

with the following form:

$$P(\Sigma):\ (\forall \varepsilon \in \Sigma:\ \widehat{P}(\varepsilon)) \tag{1}$$

where $\widehat{P}$ is a predicate on individual executions. Hence, a property is defined exclusively in terms of individual executions and may not specify a relationship between different executions of the program.

One way of enforcing security properties is with a monitor that runs in parallel with a target program [142]. Whenever a security-relevant operation is executed, the monitor checks if the operation conforms to its policy. If so, the execution sequence is allowed; otherwise the monitor modifies the execution (i.e from σ to σ') so that the property is obeyed. A program monitor can be formally modeled by a security automaton, which is a deterministic finite or countably infinite state machine [101]. This security automata introduced by Schneider has been proven capable of enforcing properties specifying that nothing bad ever happens, namely, safety properties [142].

3.3.2 Edit Automata

Edit automata [101] have been proposed as an extension that enhances security automata, so security properties can be enforced as well. An edit automaton *E* is a finite or a countably infinite state machine (Q, q_0, δ) defined with respect to some system with action set *A*, where:

- *Q* specifies the possible automaton states,
- q_0 is the initial state,

- $\delta : Q \times A \to Q \times (A \cup \{\cdot\})$ is a transition function.

$$(q,\sigma) \xrightarrow{a}_E (q',\sigma') \qquad \text{(Basic)}$$

$$\frac{\sigma = a;\sigma' \qquad \delta(q,a) = (q',a')}{(q,\sigma) \xrightarrow{a'}_E (q',\sigma)} \qquad \text{(Insertion)}$$

$$\frac{\sigma = a;\sigma' \qquad \delta(q,a) = (q',\cdot)}{(q,\sigma) \xrightarrow{\cdot}_E (q',\sigma')} \qquad \text{(Suppression)}$$

Figure 3: Edit Automata Semantics

The operational semantics of edit automata is defined in Fig. 3 where:

- a and a' denote individual actions,
- "$\cdot$" denotes empty action,
- σ and σ' denote action sequences, and
- ";" denotes concatenation of actions or action sequences (i.e. $\tau;\sigma$ denotes an action followed by action sequence). Actions emitted by the automata are indicated above the arrowed line ($\longrightarrow$).

A computation of the edit automaton is a finite or infinite sequence

$$(q_0,\sigma_0) \xrightarrow{a_1} (q_1,\sigma_1) \xrightarrow{a_2} \ldots \xrightarrow{a_n} (q_n,\sigma_n).$$

In such a computation, on the input σ_0 starting from state q_0, the edit automaton produces the output $a_1;a_2;\ldots a_n$. We define the function $\mathcal{T}_E$: $A^* \to A^*$ as the function assigning to each execution σ its output σ' from the initial state q_0. The reflexive and transitive closure of $\longrightarrow$ is denoted as $\longrightarrow_*$.

An edit automaton is powerful since it combines the power of suppression automata and insertion automata, while also being able to behave as a truncation automaton if desired. A truncation automaton would simply halt the program upon the detection of a violation. A suppression automaton can halt a program, but additionally, it is capable of suppressing individual actions so that the program does not terminate upon a violation. An insertion automaton can halt a program, but it is also capable of inserting a sequence of actions into the program action stream. Since edit automata combine these powers, they are capable of preserving the program's semantics while assuring that the security property will not be violated.

Although the edit automata model has been presented for security enforcement, we consider the model as very attractive for describing security properties. The model is capable of recognizing program sequences and distinguishing between invulnerable and vulnerable sequences; hence it is possible to take advantage of the model to specify security properties. However, edit automata are non-deterministic; if identical actions affect multiple properties, the automata's reaction is undefined. For the purpose of security testing, deterministic behavior is mandatory, and so modifications to the model are necessary prior to being able to use it for our purpose.

3.3.3 Team Automata

Team automata [53] have been introduced as a mathematical model of groupware systems. The model initially introduces component automata, and shows how multiple component automata can be interconnected to form a team; that is team automata. Teams can then be interconnected to form a larger architecture. Component automata differ from ordinary automata in the way they view actions. Actions of component automata are classified as input actions, output actions, and internal actions. These actions are used to connect multiple automata. For example, two automata can be connected by having the output action of one of them as the input action to the other. A brief description of component automata and team automata [53] is given below.

A component automaton *C* is defined as a four-tuple $\langle Q, Q_0, \Sigma_C, \delta \rangle$, where:

- *Q* is a non-empty set of states,
- Q_0 is a non-empty set of initial states such that $Q_0 \subseteq Q$,
- Σ_C is defined as (Σ_{in}, Σ_{out}, Σ_{int}), where:

  Σ_{in} defines input actions,

  Σ_{out} defines output actions, and

  Σ_{int} defines internal actions.

  The distinctions are that input actions are not under the local system's control and are caused by another non-local component, the output actions are under the system's control and externally observable by other components, and internal actions are under the local system's control but are not externally observable.

- $\delta : Q \times \Sigma_C \rightarrow Q$ is a state transition function.

For example, consider the two component automata C_1 and C_1 shown in Figure 4. The actions of these automata can be described as:

$a_1, a_1^{'}, c_1, c_1^{'} \in \Sigma_{1,int}$,

$b_1 \in \Sigma_{1,out}$,

$a_2, a_2^{'}, e_2 \in \Sigma_{2,int}$,

$b_2 \in \Sigma_{2,out}$, and

$c_2, d_2 \in \Sigma_{2,in}$.

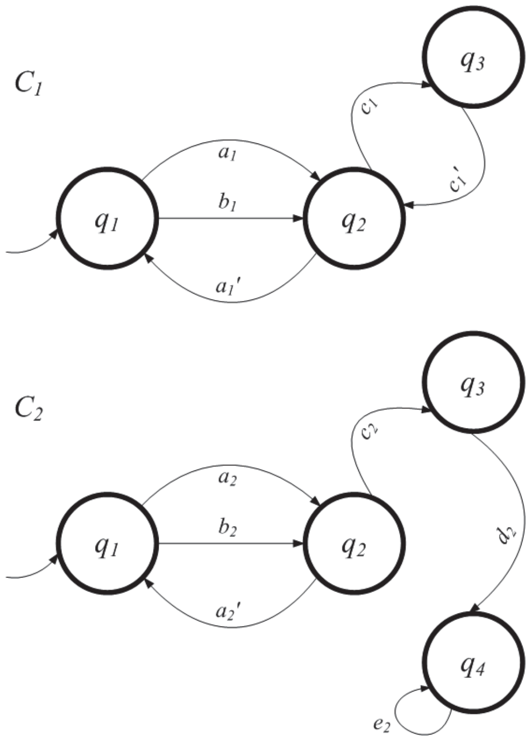


Figure 4: Component Automata Examples

Now, given a set of component automata $\{C_1, \ldots, C_n\}$, a team automaton *T* is a four-tuple $\langle Q^T, Q_0^T, \Sigma_T, \delta^T \rangle$, where:

- $Q^T = \Pi_{i=0}^{n} Q_i$ is a non-empty set of states, where Π denotes the Cartesian product,

- $Q_0^T = \Pi_{i=0}^n Q_{0i}$ is a non-empty set of initial states,
- Σ_T is an action signature defined as (Σ_{in}, Σ_{out}, Σ_{int}), where:

  $\Sigma_{out} = \bigcup_{i=0}^n \Sigma_{i,out}$, the union of all component automata output sets,

  $\Sigma_{in} = (\bigcup_{i=0}^n \Sigma_{i,in}) \setminus \Sigma_{out}$, the union of all component automata input sets minus component automata output sets,

  $\Sigma_{int} = \bigcup_{i=0}^n \Sigma_{i,int}$, the union of all component automata internal sets,
- $\delta^T : Q^T \times \Sigma_T \to Q^T$ is a transition function defined on all possible input actions to the team.

Consequently, given a single action, a team automaton is capable of finding all components in the team which can, from their current states, execute that action, and requiring all of them to execute it simultaneously. This behavior is similar to our intended approach. A team automaton can be defined as: (1) cooperating, if it is structured so that one of the automata is the active master, and all others are passive slaves; or (2) collaborating, if it is structured so that all of the automata are active peers.

3.3.4 Team Edit Automata

In this section, we present our core approach to SCSL; namely Team Edit Automata (TEA). The team edit automata model incorporates the powerful enforcing capability of edit automata into the architectural model of team automata, which allows interaction between different components of the team. Particularly, a team edit automaton is composed of one or multiple component edit automata. The team can define which flaws detected by its component automata should be reported and which ones should be suppressed. Since a team edit automaton is composed of multiple component edit automata, we first introduce component edit automata, and then follow with the description of the team itself. For consistency with similar related work, we use the same notation used for security automata and edit automata.

Component Edit Automata A component edit automaton is used to simulate program monitoring for one security property. A component edit automaton is a five-tuple $\langle Q, Q_0, \Sigma_C, G, \delta\rangle$, where:

- *Q* is a non-empty finite, or countably infinite, set of states,
- Q_0 is a non-empty set of initial states such that $Q_0 \subseteq Q$,
- Σ_C is defined as (Σ_{in}, Σ_{out}, Σ_{int}), where:

  Σ_{in} defines input actions,

  Σ_{out} defines input actions, and

  Σ_{int} defines internal actions,
- *G* is a set $\{g_1, \ldots, g_n\}$ of guard conditions,
- $\delta : Q \times \Sigma_C \times G \to Q \times (\Sigma_C \cup \{\cdot\}) \times G$ is a partial function specifying the state transition function of the automaton.

A guard condition g allows a component edit automaton to verify context information before making state transition. This increases the expressiveness of our model over edit automata, where the state transitions are only based on temporal properties of the input action.

The partial function determines the output action based on the current state of the automaton, namely, the input action and the guard condition. Based on this combination, the function determines whether the automaton should suppress the input action, report errors, or insert actions to the output. All the undefined transitions by δ are considered to be errors, which would keep the automaton in its current state and report security violation to the team. The operational semantics of component edit automata is specified in Fig. 5.

$$(q, \sigma, g) \xrightarrow{a} (q', \sigma', g') \qquad \text{(Basic)}$$

$$\frac{\sigma = a; \sigma' \qquad \delta(q, a, g) = (q', a', g')}{(q, \sigma, g) \xrightarrow{a'} (q', \sigma, g')} \qquad \text{(Insertion)}$$

$$\frac{\sigma = a; \sigma' \qquad \delta(q, a, g) = (q', ., g')}{(q, \sigma, g) \xrightarrow{\cdot} (q', \sigma', g')} \qquad \text{(Suppression)}$$

$$\frac{\sigma = a; \sigma' \qquad \delta(q, a, g) = (q, ., g)}{(q, \sigma, g) \xrightarrow{r} (q, ., g)} \qquad \text{(Report Flaw)}$$

Figure 5: Component Edit Automata Semantics

The semantics of insertion and suppression are similar to the one defined in edit automata. The "report flaw" case does not confirm that a flaw exists; rather it only reports that a flaw is possible. Upon an input action *a*, for which the transition function δ is not defined, the automaton outputs a special action *r*, standing for report-flaw. The output action *r* signals to the team that there is a potential flaw in the program.

$$(q, \sigma, g) \xrightarrow{\{\sigma_1\}}_T \{(q', \sigma', g')\} \qquad \text{(Basic)}$$

$$\frac{(q_1, \sigma, g_1) \xrightarrow{\sigma_1}_{*C_1} (q_1', \sigma', g_1') \qquad (q_2, \sigma_1, g_1') \xrightarrow{\sigma_2}_{*C_2} (q_2', \sigma'', g_2)}{(q_1, \sigma, g_1) \xrightarrow{\{\sigma_2\}}_T \{(q_2', \sigma'', g_2)\}} \qquad \text{(Pipelining)}$$

$$\frac{(q_1, \sigma, g_1) \xrightarrow{\sigma_1}_{*C_1} (q_1', \sigma', g_1') \qquad (q_2, \sigma, g_2) \xrightarrow{\sigma_2}_{*C_2} (q_2', \sigma'', g_2')}{(q_1, \sigma, g_1) \xrightarrow{\{\sigma_1, \sigma_2\}}_T \{(q_1', \sigma', g_1'), (q_2', \sigma'', g_2')\}} \qquad \text{(Concurrency)}$$

$$\frac{(q, \sigma, g) \xrightarrow{s}_T s' \qquad \sigma' \in s \qquad \sigma' = r}{(q, \sigma, g) \xrightarrow{r}_T \{(q, ., g)\}} \qquad \text{(Report Flaw)}$$

Figure 6: Team Edit Automata Semantics

Team Edit Automata It is composed of one or more correlated component edit automata connected using actions. The team controls the interaction between these automata, and ensures that all of them will respond to program inputs with explicitly defined outputs. In the same vein as team automata, given a set of component automata $\{C_1, \ldots, C_n\}$, a team edit automaton *T* is a four-tuple $\langle Q^T, Q_0^T, \Sigma_T, \delta^T \rangle$, where:

- $Q^T = \Pi_{i=0}^n Q_i$ is a non-empty set of states, where Π denotes the Cartesian product,
- $Q_0^T = \Pi_{i=0}^n Q_{0i}$ is a non-empty set of initial states,
- Σ_T is an action signature. It is composed of all single actions from any one of the component automata. Consequently, the internal action set of *T* is the union of the internal action sets of the components. The T output action set is the union of the component automata output sets whereas the T input action set is the union of all

component inputs minus the set of all component outputs. These actions sets can be defined as:

$\Sigma_{int} = \bigcup_{i=0}^{n} \Sigma_{i,int}$,

$\Sigma_{out} = \bigcup_{i=0}^{n} \Sigma_{i,out}$, and

$\Sigma_{in} = (\bigcup_{i=0}^{n} \Sigma_{i,in}) \setminus \Sigma_{out}$,

- $\delta^T : Q^T \times \Sigma_T \times G^T \to Q^T \times (\Sigma_T \cup \{\cdot\}) \times G^T$ is a transition function defined on all possible input actions to the team, where G^T is a non-empty set of guard conditions, defined as: $G^T = \Pi_{i=0}^{n} G_i$.

A team edit automaton can be defined as:

- *Pipelining team edit automata.* A team edit automata is pipelining if the output from one of its components is the input of another component. Given a single action, a team edit automaton is capable of finding all component edit automata in the team which can, from their current states, execute that action, and requiring all of them to execute it.

- *Concurrent team edit automata.* Given an output action from any component edit automaton in the team, a team edit automaton is capable of finding all component edit automata in the team which can, from their current states, execute that action, and requiring all of them to execute it.

- *Combinative team edit automata.* It contains component edit automata that have both pipelining and concurrency relationships.

The operational semantics of team edit automata is presented in Figure 6, where rule Pipelining represents the pipelining team edit automata and rule Concurrency represents the concurrent team edit. Rule Report Flaw describes the case when the team edit automata reports a flaw. In the dynamic context, this flaw represents a vulnerability, whereas in the static context, this flaw may represent a vulnerability. Final decisions should be provided by the team should flaws occur. These decisions should be specified according to the particulars of the security properties that have been checked.

Example Consider a system that enforces (1) a disable/enable-interrupt policy, (2) a privileged directory access policy, and (3) a user/process authentication policy. Only processes running on a supervisor mode should be able to access/modify/delete files in a privileged directory, or disable/enable system interrupts. In order to enforce these policies, three component edit automata are constructed for each policy to form a team. The output of the authentication automaton is piped to both automata of directory access and disable/enable interrupt. In such architecture, the authentication automaton has a pipelining relationship with the other two automata. However, both automata of directory access and disable/enable interrupt have concurrency relationships.

3.4 Design and Implementation of Team Edit Automata

In this section, we highlight the design and the implementation of Team Edit Automata (TEA). Further details of the implementation are deferred to Chapter 7.

3.4.1 Design Overview of TEA

In general, a TEA is composed of one or more component edit automata. Both the team and the components are triggered through events. Components are related to each other (belong to the same team) using actions. Should a team receives multiple outputs from different components, the team must determine the final action to be taken. In our design, the team actually refers the outputs to another component, referred to as Suggestion Solver to finally determines the action to be taken.

We utilize Object-Oriented Programming, particularly C++, for our implementation of TEA. Four main classes compose our design, namely `Event`, `ComponentAutomata`, `TeamAutomata` and `SuggestionSolver`.

- Class `Event` is provided as an abstract class, and is to be implemented by inherited concrete classes based on the particulars of the application being tested. An `Event` object is the entity that triggers both component and team automata, which may lead to a state transition of these automata.

- Class `ComponentAutomata` is provided as an abstract class. Security properties are represented as objects of this class; more particularly as objects from concrete classes inheriting that class.

- Class `TeamAutomata` relates and enables interactions between different components. The class implements the functionalities needed to allow such interaction, as well as, other functionalities needed for team management. Upon any program action, the team guarantees a proper and deterministic response to that action.

- Class `SuggestionSolver` provides the actual capability for taking final decisions should flaws occur. Our implementation provides a simple decision solver that determines the final decision based on a pre-defined set of prioritized different suggestions. However, security analysts should implement their own suggestion-solver according to the particulars of the security properties they are checking.

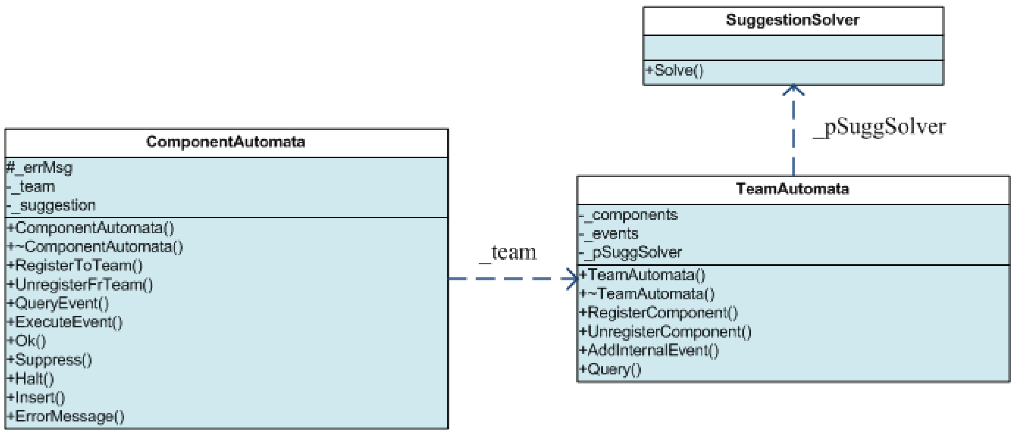


Figure 7: Partial Overview of Team Edit Automata Classes

3.4.2 Implementation of TEA

The relations between the different classes together with their main contents are illustrated in Figure 7. The `ComponentAutomata` class is provided as an abstract class. Objects from this classes are associated to a team through the `_team` private attribute. A component is capable of changing teams through the utilization of the public methods `UnregisterToTeam()` and `RegisterToTeam()`.

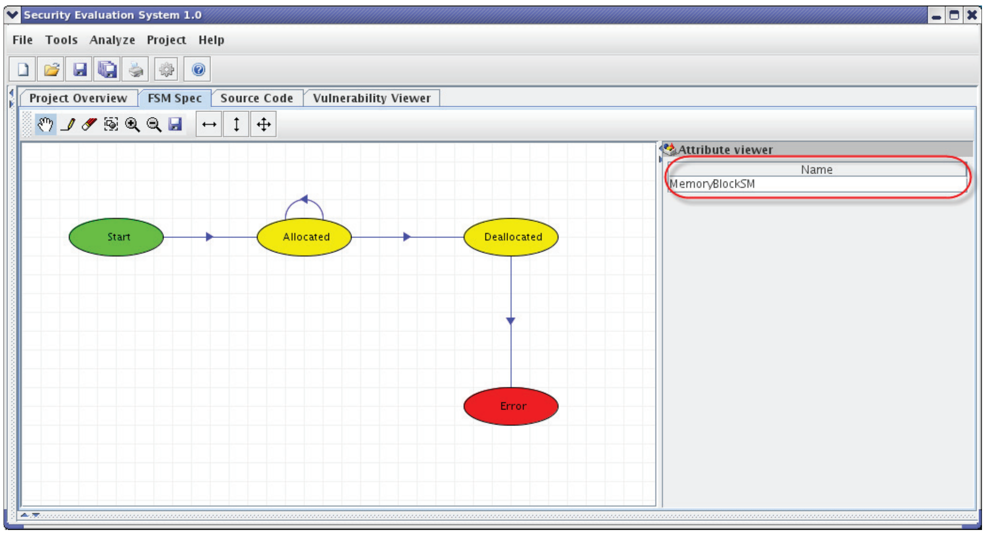
The `Ok()`, `Suppress()`, `Halt()` and `Insert()` methods are used to enforce the proper reaction of the automata. For instance, should a violation is detected, the automaton may halt program execution or suppress the violating action then continue execution. The automaton may alternatively insert an additional action, for corrective measures, before execution continues.

Upon the occurrence of an input event, the component would perform two operations through the execution of the `QueryEvent()` and `ExecuteEvent()` methods accordingly. First, the component responds to the input action with a suggestion without actually performing state transition. The actual transition is then performed through the execution of the later method.

The `TeamAutomata` class provides the implementation of the TEA. The `RegisterComponent()` and `UnregisterComponent()` methods are used to register/unregister component automata into the team. A team is capable of managing those automata that are registered to it. The class maintains a private data structure, `_events`, where it stores unprocessed events. Events from components automata are enqueued into this data structure through the utilization of the `AddInternalEvent()` method.

As a separation of concerns, and to allow security analysts to define how outputs icluding flaws will be dealt with in any way they wish, the issue of handling outputs is isolated from the team. The team refers this operation to the `SuggestionSolver`. Upon the occurrence of an input action, the `Query()` method broadcasts that action to all components of the team to obtain their suggestions. These suggestions are then sent to the `SuggestionSolver` to produce a team-wise suggestion. Security analyst should implement their own suggestion-solver according to the particulars of the application under test.

Finally, to facilitate the work of security analysts, we provide a graphical user interface facility with which security analysts can express the property to be tested by drawing the automaton instead of writing its object-oriented code. Figure 8 provides a view of our interface. More information of our graphical utility is provided in Chapter 7.



Name
MemoryBlockSM

Figure 8: Graphical User Interface for Writing Security Properties

3.5 Conclusion

In this chapter, we have introduced a mathematical model, namely Team Edit Automata (TEA). With this model, security analysts are capable of precisely specifying the security properties that they wish to test the software against. In addition to achieving the main requirement of allowing the analysts to specify the security properties, this research effort has achieved the following goals: 1) allowing the user to easily specify the desired security properties through a simple easy-to-learn graphical notation. 2) allowing the composition of several specifications of security properties. and 3) allowing the specification of mitigation measures upon the detection of a given security vulnerability. We have highlighted the design of the Team Edit Automata and provided the main details of its class hierarchy and implementation.

Chapter 4

AOP-Compiler-Assisted Code Instrumentation

4.1 Introduction

It is sometimes necessary to inject additional code at particular points in a software. The execution of the injected code can then be used for different reasons, such as providing profiling information about the software behavior, or even controlling/altering the original behavior of the software. Code instrumentation is hence considered a crucial subject in relation to various software security fields, such as security testing and security hardening. In spite of the fact that code instrumentation has the potential of disrupting the original behavior of software, and that other alternatives to code instrumentation could be used, it is likely that code instrumentation would provide the best results for such purposes. An alternative to code instrumentation is statistical sampling. With statistical sampling, individual observations are recorded then used, statistically, to form the needed knowledge.

While statistical sampling is less disruptive to program execution, it cannot provide complete accurate information [99]. For instance, statistical sampling may be able to reveal the percentage of time spent by CPU for execution of some frequently-called functions, whereas code instrumentation can provide the exact number of calls made to each of them, as well as the time spent in each call.

While the general idea of code instrumentation is common; that is just inject additional monitoring code in the base code, various code instrumentation techniques exist, as we previously discussed in Chapter 2. These techniques however carry various tradeoffs. The main contribution of the work presented in this chapter is a code instrumentation approach that is more suitable for software security testing and vulnerability detection. The approach takes special interest in Aspect-Oriented Programming (AOP) as a framework and a technology for code instrumentation. More precisely, the approach is based on both AOP and the compiler-assisted approach. In that context, we first discuss the applicability and usability of AOP for code instrumentation. A detailed introduction of our approach to code instrumentation is then presented. We finally present our extension to the GCC compiler to support our code instrumentation technique.

4.2 Aspect-Oriented Programming Instrumentation

As a part of this research effort, we investigate the applicability and usability of Aspect-Oriented Programming (AOP) [87] as a tool for code instrumentation for security testing. In this section, we provide a brief introduction of AOP, explain our interest on it as a tool for code instrumentation, and highlight its applicability and limitations for such purpose.

Following that, we highlight the needed extensions to AOP to become usable for security testing code instrumentation, and detail our contribution to that subject.

4.2.1 Aspect-Oriented Programming

Aspect Oriented Programming (AOP) [87] is a paradigm that improves software modularity by allowing encapsulation of crosscutting concerns such as security, synchronization, etc. AOP uses a weaving mechanism to merge code in a natural and systematic way. Generally, weaving is the process of composing core functionality modules with aspects into one single application. Additionally, one of the most used AOP approaches is the pointcut-advice model. The fundamental concepts of the pointcut-advice approach are: *join points*, *pointcuts* and *advices*. A join point is a point in the execution of an application. A pointcut is a constructor that designates a set of join points, and an advice is a code segment attached to specified pointcuts. Advice is executed when join points satisfying its pointcut are reached. Generally, there are three kinds of advices: *before*, *after* and *around*. AOP executes the before and after advices before and after intercepted join points, respectively. The around advice allows the possibility to either execute or skip the join point.

Other AOP approaches are available such as multi-dimensional separation of concerns [152] and adaptive programming [100]. Multi-dimensional separation of concerns allows simultaneous separation according to multiple, arbitrary kinds (dimensions) of concerns. Adaptive programming motivates specific programming style that produces as much as possible loose coupling between the program structure and the concerns.

4.2.2 AOP for Security Testing

We consider AOP, and in particular the pointcut-advice model, as a very attractive alternative for code instrumentation for the purpose of security testing. AOP allows for the separation of crosscutting concerns. Security analysts are capable of injecting monitoring code at those particular points where profiling is needed, and such injections can be done in an automatic way. Further, if the profiling code is instrumented as aspects, it can then be merged with the original code to produce final instrumented code that is suitable for the intended security testing. Another significant advantage of AOP is that many AOP languages have already been developed as language-dependent for various programming languages. For instance, AspectC [35] is built on the top of the C programming language, AspectC++ [147] is built on the top of the C++ programming language, AspectJ [86] is built on the top of the Java programming language, AspectC# [89] is built on the top of the C# programming language, and Apostle, an AOP for the Smalltalk programming language [13]. The attractiveness of AOP motivates us to elaborate a framework for security testing using the aspect-oriented paradigm. However, the current set of available pointcuts is insufficient for the purpose of security testing and hence an extension to this set is needed, as we explain in Section 4.2.3.

Since multiple AOP languages exist, we focus our research on one of them, AspectC++ [147], an AOP extension for C++. AspectC++ provides two types of pointcuts: *name pointcuts*, which describe some known entities in the program, and *code pointcuts*, which describe a collection of points in the control-flow of the program under execution.

```
1:#include <iostream>
2:#include <string>
3:using namespace std;
4:int main()
5:{
6:  char *ptr_a, *ptr_b;
7:  ptr_b = new char[200];
8:  strcpy(ptr_b, "Benign string.");
9:  cout « ptr_b « endl;
10: delete [ ] ptr_b;
11: ptr_a = ptr_b;
12: strcpy(ptr_a, "Malicious code.");
13: cout « ptr_a « endl;
14: return 0;
15:}
```

Figure 9: Illustrative Example of Current Limitations of AspectC++ Pointcuts

For instance, assume the existence of a C++ class called `SomeClass`, it is possible to match all methods in that class that return an integer through the utilization of "`int SomeClass::%(...)`" name pointcut. Describing the calls of such methods at runtime is made through the utilization of a pointcut as follows: "`call(int SomeClass::%(...))`".

While these two types of pointcuts are quite useful, they are insufficient for our intended instrumentation for security purposes. The main problem is the incapability of these pointcuts to target local and global variables in C/C++ code. Figure 9 provides an illustrative example that shows the problem. The code shown in Figure 9 is rather a simple example of a vulnerable memory manipulation. The code deletes an allocated memory at line 10 then utilizes this memory again at line 12. Although this code would compile and execute, it is clearly vulnerable since data are written to unallocated memory.

These types of errors can clearly be detected by injecting monitoring code within the method; particularly before and after the variable is being accessed. However, this is not currently possible since AspectC++ pointcuts do not support this type of injection. Consequently, if AspectC++ is meant to be used for the intended security testing purposes, a

```
<VariablePointcut>        ::=  <NameVariablePointcut>              |  <CodeVariablePointcut>
<CodeVariablePointcut>    ::=  get(<NameVariablePointcut>)        |  set(<NameVariablePointcut>)  |
                               declare(<NameVariablePointcut>)
<NameVariablePointcut>    ::=  <Type> [<NameSpace>][<Class>|<Struct>]::[<Function>]::<VariableName>
```

Figure 10: New AspectC++ Pointcuts

new set of pointcuts has to be designed and implemented. We introduce these pointcuts in Section 4.2.3.

4.2.3 New Pointcut Extension for AspectC++

In order to allow the utilization of AOP for the intended security purposes, we introduce an extension of four new pointcuts. Since we need to match local/global variables in C/C++ code, we first introduce a name pointcut that allows us to match these variable names. We then define three code pointcuts that allow us to keep track of the declaration and manipulation (read and write) of these variables. Figure 10 shows our new pointcuts.

To illustrate how the new pointcuts can be used to detect runtime errors, such as the ones shown in Figure 9, we first need to define an aspect for monitoring pointer operations. This aspect, which utilizes our new pointcuts, is shown in Figure 11. In the `Pointer-Tracker` aspect, shown in Figure 11, `MemoryTrackingList` and `PointerTrack-ingList` are our user-defined lists that keep track of allocated memory space and declared pointer variables. The `declared()` and `write()` pointcuts define pointcuts for pointer declaration and assignment respectively. The `writeToContent()` and `readToCon-tent()` pointcuts define pointcuts for writing and reading memory spaces referenced by pointers respectively. The remaining part of the aspect defines four advices that control

```
aspect PointerTracker {

 static MemoryTrackingList mtl;
 static PointerTrackingList ptl;

 pointcut declared() = declare("%* %");
 pointcut write() = set("%* %");
 pointcut writeToContent() = set("* (%* %)");
 pointcut readToContent() = get("* (%* %)");

 advice declared() :  after() {
  // register pointer variable in ptl
 }
 advice write() :  after() {
  // update pointer's right-hand value (rvalue) in ptl
  // detach pointer from old memory space
  // attach it to new memory space or NULL
 }
 advice writeToContent() :  before() {
   // report error if pointer attaches to NULL
 }
 advice readContent() :  before() {
  // report error if pointer attaches to NULL
 }
}
```

Figure 11: Utilization of the New Pointcuts for Tracking Memory Management Operations

the actions taken at the four pointcuts. Whenever a pointer is declared, the weaved code registers the pointer to the `PointerTrackingList`. A change to an allocated memory space is registered to the `MemoryTrackingList`, which keeps track of this memory as well as pointers pointing to it. All pointers not pointing to properly allocated memory space are associated with a special `NULL` record in the list.

The utilization of this aspect allows overcoming the limitation of the current AspectC++ implementation. For instance, by weaving this aspect into codes that are similar to the one shown in Figure 9, the weaved application is capable of reporting errors should an attempt to utilize a NULL pointer is detected.

4.3 AOP-Compiler-Assisted Code Instrumentation Approach

Understanding the benefits of the pointcut model of AOP, we designed and implemented a new approach, namely AOP-Compiler-Assisted Code Instrumentation. This approach takes advantages of the powerful capabilities of both AOP pointcuts and the compiler-assisted approach, described in Section 2.5.5.

The following major facts are behind our choice for combining AOP with the Compiler-Assisted approach in particular:

- Program structures are naturally known to compilers. Compilers know the lexical structure and semantics of the code being compiled. Hence, they are capable of building a more structured representation of the code (i.e. Abstract Syntax Tree (AST) and Control Flow Graphs (CFG)). Consequently, instrumentation can be performed very precisely at any program points.

- Better execution performance can be achieved as a result of utilizing compilers. This issue is crucial to dynamic analysis, where multiple executions of the program, possibly a very large number, may be needed before vulnerabilities can be detected. Since compilers perform program optimization, they are capable of optimizing the instrumented code as well. There is also another performance enhancement. Since the monitoring code is complied as part of the instrumented code, a faster executable is obtained.

Our decision to use the compiler-assisted approach in fact raises three major concerns:

1. Which compiler?

2. Is this compiler capable of supporting various programming languages?

3. Does this compiler provide what we need for security instrumentation?

The answer to the first two questions was the GNU Compiler Collection (GCC) [55]. There are many reasons behind our choice of GCC. Beside the fact that GCC is well-developed and has heavily been used and tested in multiple platforms, the compiler supports the compilation of various programming languages such as C, Objective C, C++, Java, Fortran, and more. GCC uses a universal intermediate language, called GIMPLE, to represent programs written in different languages. Consequently, by instrumenting at the GIMPLE level, support for various languages can be achieved. The answer to the third question is "no". Hence, in order to use GCC, we need to first extend it. In fact, various extensions are needed in order to support various programming languages.

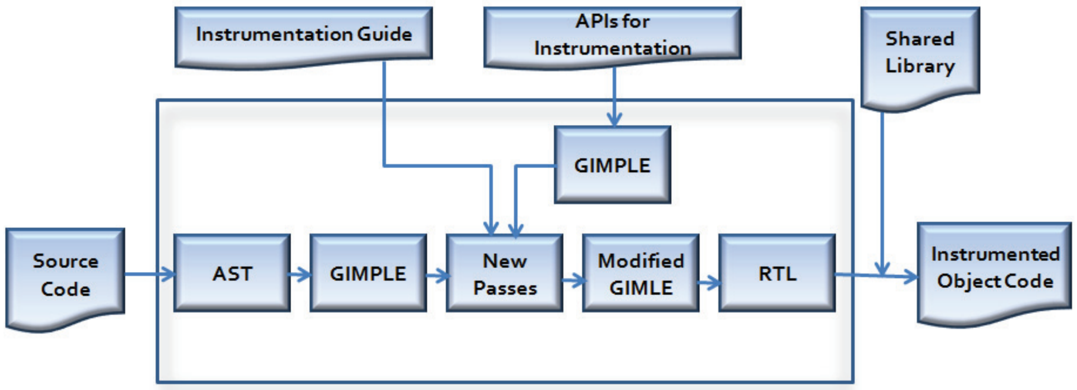


Figure 12: Workflow of the GCC Extension for Code Instrumentation

Considering the security issues that the C language is known to suffer due to its underlying implementation of data types and operations, we view the language as one that highly deserves attention, in terms of security. Consequently, we implement an extension to the GCC compiler for C. Figure 12 shows a workflow of the GCC extension for code instrumentation.

With this extension, GCC is capable of providing automatic code instrumentation. The rules of such instrumentation, i.e., what and where to instrument, is provided through an input file, referred to as the *Instrumentation Guide*. The code to be instrumented is provided as a shared library. As shown in Figure 12, three components are provided to the GCC extension: the original source code, an instrumentation guide, and the instrumentation routines, which are provided as a static library to the middle-end component of GCC. Further details of CCC internals are provided in Section 4.4.1. While adding the instrumentation routines statically; i.e. at compile/linking time, may be sufficient in many cases, it has some limitations. For instance, the instrumentation routines must be written in the same language as the one used to write the original source code, which may not be convenient in some cases. To provide further flexibility, our extension allows for the instrumentation routines to be provided as a shared library that are dynamically linked to the instrumented program. It is possible for the analyst to write the instrumentation routines in a different language than the wrap these routines into interfaces of the language used for the source code. The wrapped routines can then be supplied as part of the shared library.

The extension is capable of instrumenting code at any of the following points in a C program:

- ***Function call:*** Programs are often composed of many functions/methods that interact to provide the full functionality of the software. Hence, it may be crucial to collect profiling information at points where a method is being called. Our instrumentation allows the injection of profiling information both before and after a method is being called. Additionally, our instrumentation allows security analysts to supply arguments that are needed for the function to proceed, as well as obtaining values returned by the method after execution.

- ***Function exit:*** With this instrumentation, it is possible for security analysts to obtain profiling information at the point where the function is about to return/exit. Such instrumentation allows the tracking of intraprocedural context of a function.
- ***Variable declaration/read/write:*** This instrumentation allows the tracking of global and local variables in C/C++ programs. With such instrumentation, it is possible to detect a large spectrum of vulnerabilities caused by improper use of variables, such as illegal pointer manipulations and buffer overflow. Our instrumentation tracks variables through their names and locations in a source code and enables the tracking of all variables types in C programs.
- ***End of a variable's binding scope:*** This instrumentation is mainly needed for efficiency purposes. Local variables have a limited scope, which starts at the point they are declared and ends just before the method/block, where they are declared, returns/ends. Global variables have a longer-life starting form the point they are declared until the program, where they are declared, exits. This type of instrumentation allows the release of program monitors that keep track of a variable, once this variable goes out of scope.
- ***Pointer dereference:*** It is reasonable to state that pointers are the most powerful operators in C/C++ programs, with their capability to access, or attempt to access,

any memory space, including those out of the proper process/user-space. Hence, pointer misuse has always been one of the major techniques for security attacks. This type of instrumentation is dedicated for keeping track of pointers, which allows security analysts to monitor pointer dereference and validate pointer actions.

In addition to the instrumentation capabilities at all the above points in a program, our extension allows the injection of additional profiling code that is needed to control the program execution upon the detection of a vulnerability. The taken action is classified as suppress, insert, or halt, as mentioned in Section 3.3.4.

- ***Suppress instrumentation:*** With this type of instrumentation, it is possible for analysts to inject monitoring code at known security-sensitive points in a program for the purpose of skipping the execution of these points. There are many reasons behind such instrumentations, such as to allow program execution to continue in spite of the existence of a known vulnerability. Such execution allows the skipped vulnerability to hence become ineffective, so that further independent vulnerabilities can be detected.

- ***Insert instrumentation:*** With such instrumentation, security analysts can rather inject additional code upon the detection of a vulnerability and continue execution. Such inserted code may perform corrective handling/measures. Our extension inserts the needed routines in shared or static libraries. It is hence quite flexible for analysts to add their own defined routines to these libraries, which are then linked to the instrumented program.

- ***Halt instrumentation:*** This instrumentation allows the program execution to terminate upon the detection of a vulnerability. This is simply done through the instrumentation of code that include the `exit()` call.

4.4 Design and Implementation of GCC Code Instrumentation Extension

In this section we present our design and implementation of GCC extension in order to support code instrumentation. To facilitate the understanding of the subject, we first provide a brief background of GCC internals [57], the different phases of the compiler as well as its GENERIC and GIMPLE intermediate languages. The details of our extension is provided afterwards.

4.4.1 GCC Internals - Optimization and Code Generation Passes

GCC compilation of a source code is performed through various consecutive compilation passes. The main passes can be described as Parsing pass, Gimplification pass, Tree Static Single Assignment (SSA) pass, and Register Transfer Language (RTL) pass. These passes perform the needed conversion of text into bits, which can be optimized. Optimization is performed on both high-level and low-level representation of the code. To allow for a better modularity, GCC architecture defines three distinct logical portions where the compilation and optimization are performed, namely the Frontend, Middle-end, and Backend. GCC architecture is shown in figure 13.

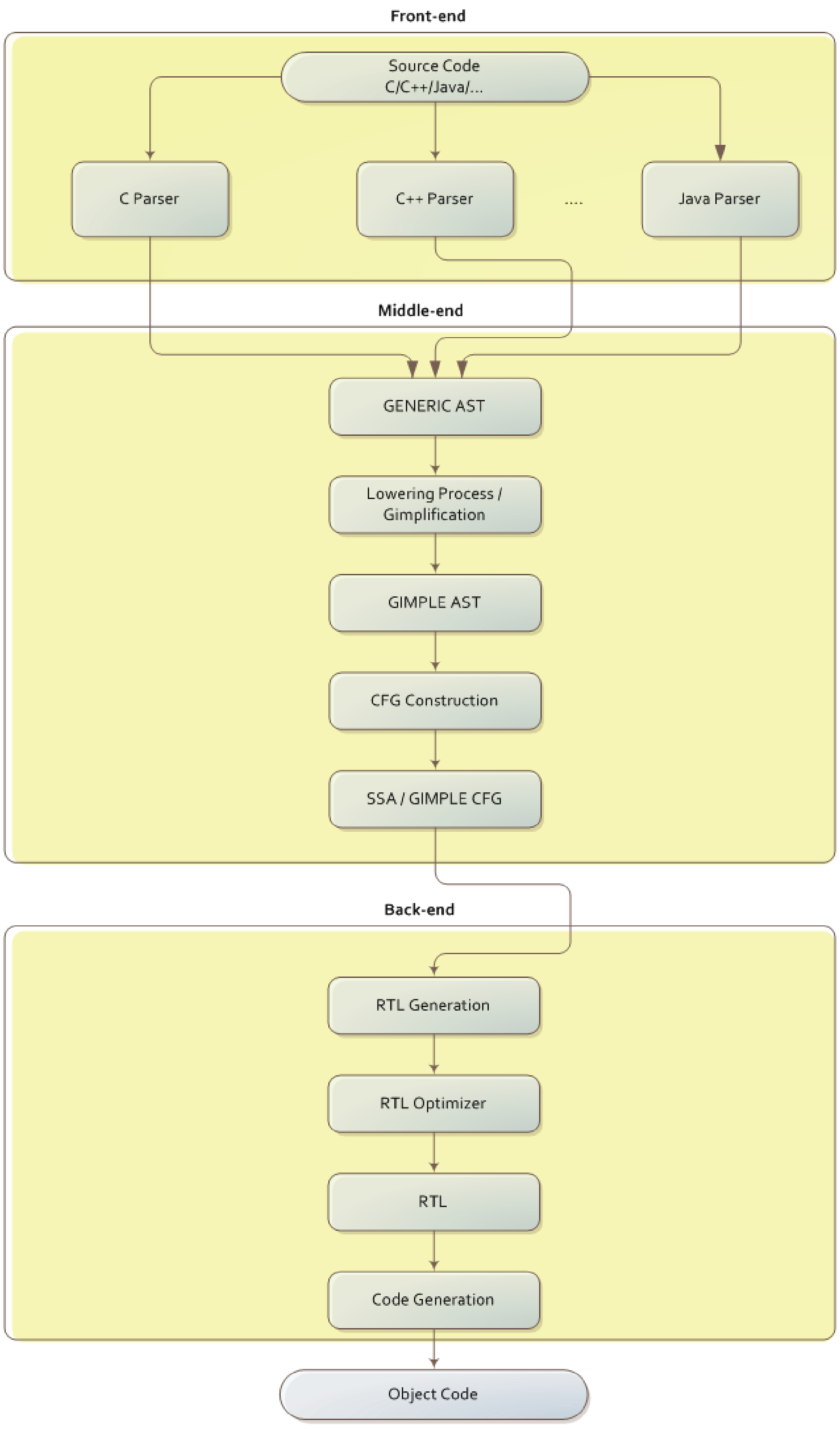


Figure 13: GCC Architecture

The source code to be compiled is accepted at the frontend. Instead of generating various language-dependent representation of the code, GCC parses the code and generates a universal language-independent output, which is referred to GENERIC Abstract Syntax Tree (AST). The GENERIC AST is a part of the middle-end portion of the architecture. However, the GENERIC representation is complex, so lowering stage as well as various further transformations are then performed at the middle-end to produce a much simpler representations. These operations result in a simpler representation, referred to as GIMPLE. Generally, nested expressions are decomposed to a three-address form where intermediate values are stored using temporary variables. Control structures, such as loops, are replaced by `gotos`. Figure 14 [57], shows an example of a GIMPLE representation for some C++ source code.

The resulted GIMPLE representation is then simplified further by transforming it into GIMPLE Control Flow Graph (CFG). These transformations take place at the middle-end part of GCC, where various optimization take place as well. The resulting GIMPLE CFG is then provided to the back-end of GCC. In that last part of the compiler, the work is done on a low-level intermediate representation called Register Transfer Language (RTL). Hence, the back-end performs further transformation of the GIMPLE into RTL, where finally the code generation takes place.

4.4.2 Extending GCC for Code Instrumentation

GCC performs its operations as a sequence of passes. These passes are controlled by a specific component referred to as the pass manager. The pass manager runs all of the

```
struct A { A(); ~A(); };

int i;
int g();
void f()
{
  A a;
  int j = (--i, i ? 0 : 1);

  for (int x = 42; x > 0; --x)
  {
    i += g()*4 + 32;
  }
}









































//end of code
```

(a) C++ Code

```
void f()
{
  int i.0, T.1, iftmp.2;
  int T.3, T.4, T.5, T.6;

  {
    struct A a;
    int j;

    __comp_ctor (&a);
    try {
      i.0 = i;
      T.1 = i.0 - 1;
      i = T.1;
      i.0 = i;
      if (i.0 == 0) iftmp.2 = 1;
      else iftmp.2 = 0;
      j = iftmp.2;
      {
        int x;

        x = 42;
        goto test;
        loop:;

        T.3 = g ();
        T.4 = T.3 * 4;
        i.0 = i;
        T.5 = T.4 + i.0;
        T.6 = T.5 + 32;
        i = T.6;
        x = x - 1;

        test:;
        if (x > 0) goto loop;
        else goto break_;
        break_:;
      }
    }
    finally {
        __comp_dtor (&a);
    }
  }
} // end of code
```

(b) GIMPLE Representation

Figure 14: GIMPLE Representation for a Sample C++ Source Code

individual passes in the correct order, and takes care of standard bookkeeping that applies to every pass. The theory of operation is that each pass defines a structure that represents everything of that pass, such as when and how it should run, and what form of intermediate language it requires. The passes are hence registered to run in some particular order. This order of execution is guaranteed to be followed by the pass manager.

Consequently, for our extension to become effective, we first need to define our new passes, register them to the pass manager and add the functionality needed by such extensions to these passes.

Defining Code Instrumentation Extension Passes For our code instrumentation extension, we define two new passes. The structure representing each of the different passes of GCC is called `tree_opt_pass`. Figure 15 shows the structure of the first pass of our extension. The details of our passes are provided next.

```
struct tree_opt_pass pass_tree_security_instrument_vardecl =
{
  "sintrument_vardecl",                 /* name */
  gate_tree_security_instrument_vardecl,/* gate */
  tree_security_instrument_vardecl,     /* execute */
  NULL,                                 /* sub */
  NULL,                                 /* next */
  0,                                    /* static_pass_number */
  0,                                    /* tv_id */
  PROP_gimple_any,                      /* properties_required */
  0,                                    /* properties_provided */
  0,                                    /* properties_destroyed */
  0,                                    /* todo_flags_start */
  TODO_Dump_func,                       /* todo_flags_finish */
  0                                     /* letter */
};
```

Figure 15: Structure of First Pass for Code Instrumentation Extension

Pass Registration The pass manager is located in three files in GCC, passes.c, tree-optimize.c and tree-pass.h. The new pass is registered to the pass manager by including it in the init_optimization_passes() method in the passes.c file.

Figure 16 shows an extract of the init_optimization_passes() method, which indicates the location of our new extension passes in the method.

```
void init_optimization_passes (void)
{
  struct tree_opt_pass **p;

  ...

  /**
   * security instrumentation phase 1
   */
  NEXT_PASS (pass_tree_security_instrument_vardecl);

  NEXT_PASS (pass_lower_omp);
  NEXT_PASS (pass_lower_cf);
  NEXT_PASS (pass_lower_eh);
  NEXT_PASS (pass_build_cfg);

  /**
   * security instrumentation phase 2
   */
  NEXT_PASS (pass_tree_security_instrument);

  ...
```

Figure 16: Registering Code Instrumentation Passes

Execution Functionality of Extension Passes The functionality of our passes is defined in the execution part of the init_optimization_passes() method. For instance, as shown if figure 15, the functionality of our first pass is defined in the tree_security_instrument_vardecl method.

Scope-Wise Code Instrumentation Extension: Pass 1

For security testing purposes, code instrumentation is required at sensitive program points, such as function calls, variable read/write, and pointer de-referencing. The first pass of our extension targets some of these points. The first three instrumentation points shown in Table 1, which are particularly related to program scope, are performed by pass one of our extension. Referring back to Figure 13, our instrumentation takes place in the middle-end part of GCC. In particular, at the higher-level GIMPLE representation before lowering. As part of the lowering operation, GCC moves all variables out of their binding context so important information needed for our instrumentation is lost.

#	Instrumentation Point	Instrumentation Pass
1	function return	Pass 1
2	variable declaration	Pass 1
3	end of variable's binding scope	Pass 1
4	function call	Pass 2
5	variable read	Pass 2
6	variable write	Pass 2
7	pointer dereference	Pass 2

Table 1: Instrumentation Points of Extension Passes

Figure 17 shows the definition of the `tree_security_instrument_vardecl` method, which provides the instrumentation functionality of Pass one. This method makes two calls to `secinstr_xform_decls` and `secinstr_xform_out_of_scope`, which instrument the code at variable declarations and end of binding scopes respectively. The `walk_tree_without_duplicates` method traverses the chain of the GIMPLE tree nodes keeping track of program scopes. While traversing, the method performs the needed instrumentation as instructed through the instrumentation guide.

```
static unsigned int tree_security_instrument_vardecl(void)
{
  int ret = 0;

  if (DECL_ARTIFICIAL (current_function_decl))
    return 0;

  push_gimplify_context ();

  secinstr_xform_decls (DECL_SAVED_TREE (current_function_decl),
                        DECL_ARGUMENTS (current_function_decl));

  secinstr_xform_out_of_scope (DECL_SAVED_TREE (current_function_decl),
                               DECL_ARGUMENTS (current_function_decl));

  pop_gimplify_context (NULL);

  return ret;
}

static void secinstr_xform_decls (tree fnbody, tree fnparams)
{
  struct secinstr_decls_data d;
  d.param_decls = fnparams;
  walk_tree_without_duplicates (&fnbody, secinstr_xfn_xform_decls, &d);
}

static void secinstr_xform_out_of_scope (tree fnbody, tree fnparams)
{
  struct secinstr_decls_data d;
  d.param_decls = fnparams;
  walk_tree_without_duplicates
    (&fnbody, secinstr_xfn_xform_out_of_scope, &d);
}
```

Figure 17: Functionality of Instrumentation Pass One

CFG-Based Code Instrumentation Extension: Pass 2

Pass Two of our extension target instrumentations in relation to function calls and variable use. The instrumentation is done through the traversal of the GIMPLE CFG for each method, and matching function calls and variable use as instructed by the instrumentation guide. If a pattern match is found, the instrumentation is performed according to the analyst instructions in the instrumentation guide.

The last four entries in Table 1 provide the instrumentation performed by Pass two of the extension. The following provides a brief explanation of the way we implement the matching for the required program points.

Matching Function Calls: Function calls are represented by GCC GIMPLE as particular tree nodes referred to as `CALL_EXPR`. The `CALL_EXPR` nodes can appear in a GIMPLE tree either as separate statement, or as the right-hand operand of an assignment statement, which GIMPLE represents through a tree node called `MODIFY_EXPR`. Our implementation hence searches each statement and each right-hand operand for these nodes. Once a match is found for a `CALL_EXPR`, we need to obtain the actual name of the calling method to compare it with the ones provided by the instrumenting guide. We obtain the method name thorough the utilization of the following API:
`lang_hooks.decl_printable_name(CALL_EXPR_typed_tree, 2).`

Matching Variable Read/Write: GIMPLE trees are stated in three-address form. Hence, our implementation searches the left-hand side of assignment statements to identify those variables that can be written. Similarly, searching the right-hand side of the statements identify those variables that are only read. The two searches precisely identify the correct type of variable use in the program.

Matching pointer de-referencing: In a similar fashion to the above matches, we search those specific nodes in the GIMPLE tree that are related to pointer de-referencing. Those nodes are called `INDIRECT_REF`. The `INDIRECT_REF` are allowed by GIMPLE to appear as right-hand or left-hand operands of an assignment statement. Our search targets both for matching. We additionally keep track on whether the de-referencing was read or write depending on its location on the statement. This differentiation is insignificant in terms of the required matching, but it can be used for future extensions if needed.

4.4.3 Execution of GCC Extension

Security analysts can utilize our GCC extension for code instrumentation either through the utilization of command line option, or by defining environment variables.

Command Line Option Security analysts can execute GCC extension by including a particular command line option when initiating GCC, and additionally indicating specific linking options to specific libraries. The command line both enables the instrumentation functionality, as well as determines the specific instrumentation guide to be used. The details of the instrumentation guide is given later in this section. Typically, this option is as follows:

```
-ftree-security-instrument=instrumentation_guide_input_file
```

The "`-ftree-security-instrument`" portion of the command results in including our security instrumentation routines. The value "`instrumentation_guide_input_file`" determines which input file to be used as the instrumentation guide.

Security analysts additionally need to add specific linking options, which specify the library containing the needed program monitoring routines. This is done through the utilization of the usual GCC linking command options "`-l`" and "`-L`". A typical command line option to initiate our GCC extension would hence look as follows:

```
gcc -L/usr/lib/SecInstrShrdLib -lSecInstrShrdLib
-ftree-security-instrument=instrumentation_guide_input_file
instr_guide.c
```

Defining Environment Variables Option The command line option is very suitable for compiling one, or few, source files. For building larger projects, which is generally the expected case, a better utility, such as MAKE [56], is to be used. However writing Makefiles requires some manual intervention and effort by the analysts, who also must have the knowledge of writing such files. Since our goal is to allow for more automation, we provide this option, which eliminates the need to modify the original Makefile. To enable the instrumentation functionality, we need to provide the instrumentation guide, as well as the shared library including the instrumentation functionality. Defining environment variables allows us to do so with a minimal effort by the analysts. Typically, the analyst needs to perform three commands that are similar to the following:

1. `export FTREE_SECURITY_INSTRUMENT=`
   `/usr/FOSS/instr_guide.c`
2. `export SHARED_LIB_NAME=/usr/lib/SecInstrShrdLib`
3. `make`

The `export FTREE_SECURITY_INSTRUMENT` enables the analyst to indicate the instrumentation guide to be used. The `export SHARED_LIB_NAME` enables the setting of the code instrumentation shared library to be used. Finally, the call to `make` is needed to build our new extended version of the compiler.

GCC Extension - Instrumentation Guide Depending on the given security property and the software to be tested, various instrumentations are needed at specific program points. Hence, the exact rules of the instrumentation and the locations where it should take place must precisely be given. To allow this, an instrumentation guide is provided, with which the security analyst can supply the instrumentation rules. This information is supplied for an efficient processing by the compiler to take place. Typically, the instrumentation guide includes various instruction lines, where each is composed of eight different fields. Since these instructions are not that easy to read or construct, we refer to the guide as low-level instrumentation guide.

Low-level Instrumentation Guide Each instrumentation instruction is composed of eight ordered different fields. Table 2 provides the details of these fields. Hence, a low-level instrumentation instruction may look as follows:

```
fun1 1 4 program1.c::1843::p SecInstr_PtrDeref 1 0 0
```

Such statement would instruct our extension to performs an instrumentation of a function called `SecInstr_PtrDeref` after the de-referencing of a pointer called `p` at line number 1843 in a method called `fun1` in `program1.c`. Additionally, the instruction indicates that the `SecInstr_PtrDeref` method is returning an integer value, and that both the return value from the function and the passed argument to the function are of no use in this case, and hence they are not to be exposed.

Order	Purpose	Format and Value
1	Scope of the concerned program point	"function_name" for a function scope "*" for any scope
2	Instrumentation position	0 for instrumenting before a program point 1 for instrumenting after a program point
3	Program point for instrumentation	0 for instrumenting at function call 1 for instrumenting at variable read 2 for instrumenting at variable write 4 for instrumenting at pointer dereference 8 for instrumenting at function return 16 for instrumenting at variable declaration 32 for instrumenting at end of variable's binding scope
4	Name of a concerned variable or function	"function name" for a function call or "*" for any functions "Filename::lineNum::VariableName" for a variable name, where Filename and VariableName are strings and can both use wild character "*" and lineNume is a positive integer with 0 representing all line numbers of a source file
5	Name of the function to be instrumented in the program	a string of the function name
6	Return type of the instrumented function	0 for void type 1 for int type
7	Determine whether return value of a function call should be exposed	0 for not exposing the value 1 for exposing the value
8	Determine whether the arguments of a function call should be exposed	0 for not exposing the arguments 1 for exposing the arguments

Table 2: Fields of the Instrumentation Guide Input File

High-level Instrumentation Guide While our extension requires such low-level instrumentation instruction to process the needed instrumentation, these instructions may fairly be viewed as not being that user-friendly. To alleviate the problem, we implement a small language with which the analyst is capable of stating the instrumentation instructions in a simpler way. We refer to such instructions as high-level instrumentation instructions. The above low-level instrumentation instructions are stated as high-level instructions as follows:

```
after derefptr (program1.c::1843::p) inject "SecInstr_PtrDeref";
```

The high-level instructions are compiled into low-level instructions suitable for processing by our extension. The grammar of our high-level langauge is provided in Appendix A.

4.5 Conclusion

In this chapter, we have presented our approach to code instrumentation for security testing, which is based on Aspect-Oriented Programming (AOP) and the compiler-assisted instrumentation approach. This code instrumentation is crucial since it allows for the required instrumentation to obtain profiling information of the software being tested as well as for test-data generation as we explain in Chapter 5 and Chapter 6.

There are three major folds to the contribution presented in this chapter. Firstly, we have explored the current limitations of AOP for such security instrumentation purposes, and have proposed the needed extensions to AOP to support such functionality. In particular, we have proposed four new pointcuts to the pointcut model of AspectC++, with which the needed security instrumentation can be achieved. Secondly, we have presented our novel approach to code instrumentation, namely AOP-compiler-assisted code instrumentation. Thirdly, we have presented our extension to the GCC compiler, as well as our implementation of such extension, to support our new code instrumentation approach.

Chapter 5

Test-Data Generation through Reachability Analysis

5.1 Introduction

There is a wide spectrum of software security vulnerabilities. Such vulnerabilities can be classified in different ways. For instance, classification can be made based on the level of harmfulness, the way they can be manifested, the difficulty level to detect them, etc. Particularly, in relation to detection, one classification can be made based on the matter of reachability, where the execution of some, possibly orderly, events is sufficient to violate a security property. In that context, we refer to those properties as *reachability properties*.

Another important matter that need to be examined is in relation to the analysis used for detection. In particular, we need to examine whether, or not, static analysis can be sufficient for vulnerability detection. The intent of this chapter is to elaborate on both of these two concerns.

We first define the matter of reachability, its details and applicability. Considering such reachability analysis, we then examine the capability of sole static analysis for vulnerability detection and elaborate a new approach to test-data generation for reachability. The work performed to reach our objectives results in three main contributions:

- The proposal of reachability analysis, which can be applied for detecting reachability security properties;
- A static analyzer that automatically identifies vulnerable points in a control-flow graph;
- Test-data generator based on program slicing and pushdown automata model-checking [88].

As a summary, the chapter provides a framework for security testing against violations to reachability properties. The rest of this chapter is organized as follows. In Section 5.2, we detail our analysis of reachability and our classification of security vulnerabilities. In Section 5.3, we illustrate our approach for static generation of test-data, and provide the details of our static analyzer, namely the Static Vulnerability Revealer (SVR). Section 5.4 provides the details of our test-data generator, which is based on Moped checker [88]. Finally, in Section 6.5, we provide a conclusion of the contributions and the work presented in this chapter.

5.2 Reachability Analysis

In order to prove that there is a security violation of a property, we need to generate data with which the vulnerable code violating this property can be executed. In that sense, a vulnerable piece of code that is not reachable is actually harmless, since it cannot be executed. While it is true that concrete data needs to be generated to execute the vulnerable code, we need to clearly distinguish two different types of data-generation that might be needed for that purpose.

The first type of data generation is related to the controlling variables and parameters along the execution path to the potentially vulnerable site of the code. With such data, program execution can be driven to the site in question. However, reaching this execution point does not really mean that the violation will be triggered. For this, data generation for those variables and parameters that are controlling the behavior of the potentially vulnerable site is needed. These two sets of data could be different. Consequently, in such cases, we need to generate data twice before we can prove that any violation exists. The first data generation is needed to reach to the potentially vulnerable site, while the second generation is needed to successfully execute the violation. To illustrate the matter, let us consider the code sample shown in Figure 18.

As shown in Figure 18, the code creates an array with an allocated size in line 4 then sets the values of a number of elements of the array buffer in line 11 and line 12. However, the initialization only takes place based on specific combinations of the values of `d`, `x1`, and `x2` as shown in line 10. The code clearly has a potentially vulnerable segment spanning lines 11 and 12. Now to prove that this code is indeed vulnerable, a set of data generation must

```
...

int main (int argc, char *argv[])
{
    double d;
    int x, y, size;
    // d, x, y & size are intialized below
    ...
    int A[size];
    // Initialize the array
    for(int i = 0; i < size; i++)
    {
      A[i] = -1;
    }

    printf("Please enter the number of elements to place in the
       buffer: \n");
    int num;
    // num is initialized below, possibly from a file
    ...

    // Follow some business rules before setting the
    //  array value to the proper values
    if(d > 9876.54 && x1 > 999999 && x2 < 700000){
        for(int i = 0; i < num; i++)
      {
         A[i] = i * 10; // Potential buffer-overflow
        }
    }else{
       printf("Invalid vlaues; cannot initialize buffer.\n");
       exit(1);
    }
    return 0;
}
```

Figure 18: Sample Code for the Generation of Different Test-data Sets

first be possible for the following variables: `d`, `x1`, and `x2`. Having such generated values is necessary but yet insufficient. In such case, the generation of values for the variables `size` and `num` is needed to prove that the vulnerability indeed exists.

Consequently, in such similar cases, a first set of data is to be generated to merely reach the vulnerable site. Should the segment be reachable, a completely different set of generation may then be required depending on the internals of such code segment and the variables that control its operations. It should also be noted that, in such cases, the generation of a particular first set may not be that crucial, since that set is possibly substitutable.

For instance, if there are multiple paths in the program to the potentially vulnerable site, then the generation of one set of data to reach this point is sufficient to start attempting the second generation.

However, there are many other cases, where the mere creation of the first set is sufficient to prove a violation. This is often the case through the execution of a specific path(s), where the sequence of events performed at these paths directly triggers the vulnerability.

Assume a system that defines the following security property: "*Files must be encrypted before being saved*". The sample code shown in Figure 19 provides an illustration of a reachability property.

```
#include <stdio.h>
#include <stdlib.h>
...
void encrypt(char * buffer, char *key);
void save(FILE *outfile, char *buffer);

int main (int argc, char *argv[])
{
   ...
1   printf("Key required; please enter the value of the
   encryption key: \n");
2   int keyval;
3   scanf("%d", &keyval);
4   if(keyval > 9999){
5      if(keyval > 999999){
6         printf("Invalid key.\n")
       }else{
7         char *key = (char *)malloc(MAX_LEN);
8         memset(key, '\0', MAX_LEN);
9         snprintf(key, MAX_LEN, "%d", keyval);
10        encrypt(buffer, key);
       }
    }else{
11     printf("Invalid key.\n");
12     exit(1);
    }
13  save(outfile, buffer);
14  printf("File %s has been encrypted & saved.\n", argv[1]);
    ...
15  return 0;
}
```

Figure 19: Illustrative Sample Code for Reachability Properties

In many cases, the code shown in Figure 19 is capable of enforcing the property. However, it will be directly violating the property if the value of `keyval` results in the execution of line 6, which is then followed by line 13. Under such particular value and execution path, the file will be saved without being encrypted, violating the property under concern. It is clear that in such cases, only one set of data generation is sufficient to prove the violation. This set would include the controlling variables along the vulnerable path. Should it is possible to generate concrete values for that set, the violation is directly proven. Many of the security properties listed by the security coding rules of the United States Department of Homeland Security [45] are reachability properties. We design and implement a static-analysis based test-data generator for the purpose of producing test-data needed to detect violations to reachability properties. The details of this generation are indicated in Sections 5.3 and 5.4. However, prior to going through such details, we will first explore the matter of reachability further. In particular, we need to provide an answer to the following question: Is it possible to reduce/transform some non-reachability properties to reachability ones? The answer to this question is important since it may significantly broaden the range of violations that can be detected through reachability analysis.

Reduction to Reachability

To reach a specific point in a program, proper data, with which the execution is driven towards such a point, is to be generated. The data generation can hence be viewed as a set of constraints that must be satisfied along the execution path. For instance, to reach the potentially vulnerable point located at line number 12 in Figure 18, the set of execution constraints can be specified as: {`d > 9876.54, x1 > 999999, x2 < 700000`}.

The vulnerable point is reached if it is possible to generate data that satisfies that set. However, this does not yet prove any violation. To prove that a violation indeed exists, it is needed to generate a different set of data in relation to `num` and `size`. Consequently, these types of vulnerability are not reachability ones.

However, in many cases, it is possible to reduce non-reachability properties into reachability ones by adding additional constraints along the path that will prove the violation should the vulnerable point be reached. For instance, it is possible to prove the vulnerability shown in Figure 18 through the generation of a single set of data that satisfy the following new set of execution constraints: {`d > 9876.54, x1 > 999999, x2 < 700000, num >= size`}. Consequently, the problem at hand is reduced to a reachability one.

There are several non-reachability properties that can be reduced into reachability ones. Examples of such include vulnerabilities related to buffer overflow, numeric-overflow, numeric-underflow and division-by-zero. Hence, our reachability analysis can be used to detect a larger spectrum of vulnerabilities by reducing such non-reachability properties. For instance, looking at each of the 174 security vulnerabilities reported by the United States Department of Homeland Security [45], we find that 71 of these vulnerabilities are related to buffer-overflow and hence can be reduced to reachability analysis.

Consequently, our approach to reachability is capable of detecting a large number of vulnerabilities that are either naturally reachability ones, or non-reachability that can be reduced to reachability properties. This has led to further contributions, and in particular, our research is then extended towards the creation of an entire framework that targets the detection of reachability properties. Section 5.3 describes the components of our vulnerability detection framework, namely Static Vulnerability Revealer (SVR). Section 5.4 describes our approach to test-data generation for reachability analysis. Both of these components are heavily based on static analysis, which we investigate carefully in Section 5.5 and reexamine it again in further details in Chapter 6.

5.3 Static Vulnerability Revealer

In this section, we introduce one of the main components of our model, namely the Static Vulnerability Revealer (SVR). Given a source code of a program, and a formally specified security property, SVR main concern is to find out all 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.

Static analysis takes advantage of control-flow, data-flow, and type information generated by the compiler to construct program models that could be used to predict undesirable behaviors. Model-checking techniques excel in the efficient exploration of program models for the purpose of matching them with respect to a logical or a behavioral specification. In this research, we use a static analysis technique to automate the model construction process, and we use model-checking to explore these models. As such, we establish a synergy between these two techniques.

SVR targets the detection of security vulnerabilities in software source code. In order to provide multi-language support, we base our system on the GCC compiler [55]. Starting from version 4, GCC mainline includes the Tree-SSA [122] framework that facilitates static analysis with a universal intermediate representation (GIMPLE) common to all supported languages. SVR works on the GIMPLE representation and abstracts required information to construct program models.

Another component utilized in SVR is a conventional pushdown system model-checker called Moped [4,88], which comes with a procedural language, Remopla, for model specification. Moped performs reachability analysis of a specific statement in the Remopla code. SVR utilizes the reachability analysis capability for the purpose of verifying security properties. In our system, a security property is modeled as a Team Edit Automaton (TEA), specifying the erroneous behavior using sequences of program actions. An `error` state is introduced to represent the risk state for each automaton. The automaton is then translated into Remopla representation. During verification, program actions would trigger the state changing of the automaton. If the `error` state is reached, a sequence of program actions violating the given property is then detected. In other words, SVR converts the security detection problem to a reachability problem. However, since not all security properties can be reduced to reachability ones, further work is needed to validate those non-reachability properties.

Figure 20 gives an overall view of the SVR system, which integrates the aforementioned components. As shown in Figure 20, different phases compose our SVR system. The first phase addresses the property specification and its model extraction. To ease the work done by the security analyst, we provide a graphical capability with which the security property can be stated. Each property is specified in TEA. Given the graphical representation, our tool automatically translates the specified properties into a Remopla specification, which is then used as part of the input to the Moped model-checker. The output of this phase is a

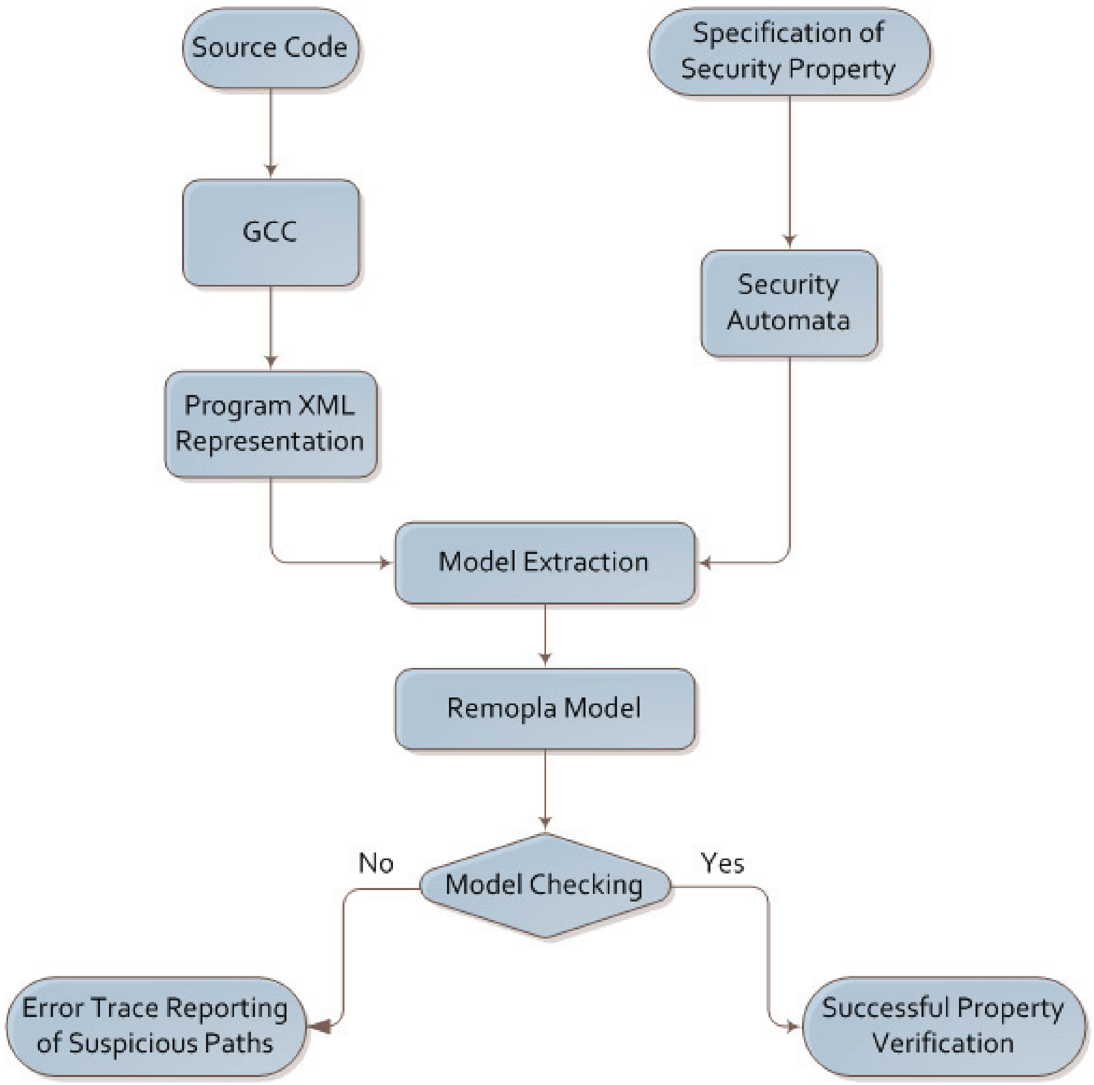

Source Code
Specification of Security Property
GCC
Security Automata
Program XML Representation
Model Extraction
Remopla Model
No
Model Checking
Yes
Error Trace Reporting of Suspicious Paths
Successful Property Verification

Figure 20: SVR System Architecture

Rempola model representing the given property.

The second phase addresses program model extraction. The input to this phase is the source code of the program under test. The output is a Remopla model that synchronizes the Remopla representation of the program with the Rempola representation of the security property. Specifically, a pass is added to the GCC compiler where the GIMPLE representation is dumped into XML files, from which the program model is extracted and represented using Remopla. The generated program model is then combined with the Rempola representation of the property. The combination is done for the purpose of: (1) achieving synchronization between the program pushdown system and the security automata, (2) binding the pattern variables of security automata with actual values taken from the source code, and (3) reducing the size of the program's model by only considering program actions that are relevant to the specified security properties.

In the third phase of our system, the resulting Remopla model from previous phases is provided as input to the Moped model-checker for reachability analysis. An error is reported when a security automaton specified in the model reaches an error state. The output of this phase is a set of suspicious paths that would possibly violate the security property in concern. It should be clearly noted that, due to the static analysis nature of SVR, some of the reported vulnerabilities could be false positives, an issue that we revisit shortly. Generally, the model-checking is the ultimate step of the SVR process.

5.3.1 Constructing Remopla Models

This subsection describes the construction of Remopla formal model in details, including the generation of the Remopla representation for both security properties and the program being analyzed.

Modeling Security Properties The security property is specified using TEA, and we focus on temporal properties. A `start` node and an `error` node are introduced, respectively, to represent the initial and the final state of a TEA. Notice that if a temporal property defines a correct behavior, the property is modeled by representing its negation. Hence, the `error` state is the risky state.

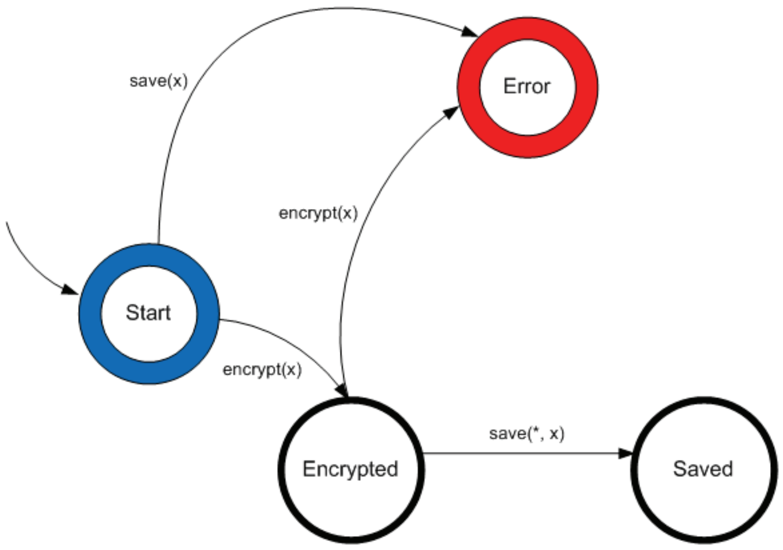


Figure 21: Security Property Specifying File Encryption Prior to Saving

From Security Automata to Remopla Given a property automaton, we serialize it into Remopla representation, which we also refer to as Remopla automaton. The latter is represented using a Remopla module. As an exmaple, Figure 22 shows the Remopla module of the file-encrypt-save security automaton in Figure 21. The nodes and the transition labels of a security automaton are mapped to Remopla constructs, as defined hereafter:

- Integers are used to identify the automaton nodes, each of which corresponds to an enumerator of a Remopla enumerated type (i.e. the enumeration variable `states` in Figure 22). For tracking the state of the automaton, an integer variable (i.e. `current_state` in Figure 22) is introduced and initialized to the automaton's initial state (i.e. `start` in Figure 22) using the Remopla keyword `INITIALIZATION`.

- Transition labels are used to identify security relevant program actions. Table 3 shows the program actions we capture, together with the corresponding Remopla representation. The Remopla constructs prefixed with ACTION_ are defined as elements of a Remopla enumeration type, which includes the relevant program actions. A transition is triggered if its label matches the input program action. A transition with the label in the first entry, for instance, is activated when the program action matches ACTION_PROGRAM_START (i.e. the main entry of a program). The second row represents the termination of the program execution, and the next two entries denote respectively the entry point of a function and its return. The mapping for a function call is defined in the fifth entry. Each function argument is an element of a global Remopla array ARG[], which is inquired during the model-checking process when function parameters are involved in the property verification. Notice that the mapping in the last entry, representing an assignment, is focused more on the program action and not on the data value being passed.

The translation from a security automaton to its Remopla representation follows the mapping defined in Table 3, and each security automaton is represented as a Remopla module. Figure 22, for instance, shows an example of such representation. The module takes the current program action as input, checks it against the defined transitions, and changes the automaton state accordingly. For example, lines 8 to 14 represent the start node of the property in Figure 21. If the action matches ACTION_FUNCTION_CALL_encrypt with parameter ARG_X, the state is changed to encrypted. A violation of the given property is detected should the error state is reached.

```
enum states{start, encrypted, saved, error};
int current_state;
INITIALIZATION: current_state = start;

move_state (int action)
{
if
  :: current_state == start ->
      if
      :: action == ACTION_FUNCTION_CALL_encrypt
          && ARG[0] == ARG_X -> current_state = encrypted;
      :: action == ACTION_FUNCTION_CALL_encrypt ->
         current_state = error;
      :: else -> break;
      fi;
  :: current_state == encrypted ->
      if
      :: action == ACTION_FUNCTION_CALL_save
          && ARG[1] == ARG_X -> current_state = saved;
      :: else -> break;
      fi;
  :: current_state == saved -> break;
  :: current_state == error -> break;
  :: else -> break;
  fi;
}
```

Figure 22: Generic Remopla Representation of the Automaton in Figure 21

Program Model Extraction The model extraction is the process that translates program source code to Remopla representation. The translation consists of two phases: (1) The GIMPLE representation of parsed source code is converted and dumped into XML files, and (2) Remopla model representing the source code is extracted from the XML files. To reduce the size of the extracted program model, a preprocessing phase is incorporated before model extraction. Since the set of properties to be verified has been specified, we have the knowledge of a set of security-related functions. By analyzing the call-graph of the program in concern, we are able to identify functions that are relevant to the verification. The size of the extracted model can hence be smaller, since it only preserves the security-relevant behavior.

Program Action	**Remopla Representation**
Program entry	`ACTION_PROGRAM_START`
Program exit	`ACTION_PROGRAM_END`
Entry of `f`	`ACTION_FUNCTION_CALL_f`
Return of `f`	`ACTION_FUNCTION_RETURN_f`
`f(`v_0`,...,`v_n`);`	`ARG[0]=ARG_`v_0`;...;ARG[n]=ARG_`v_n`; f( );`
`var =` v`;`	`ACTION_VAR_MODIFICATION_var;`

Table 3: Remopla Representation of Program Actions

The translation from program constructs to Remopla representation follows the mapping defined in Table 4. The first entry of Table 4 shows the Remopla construct for the control-flow structure in source code. Note that each condition in the source code is represented using the Remopla keyword `true`. With such a condition, the model checker would choose either branch non-deterministically during verification, considering both branches are feasible. At this stage, we take into account all paths in the source code without pruning infeasible paths, which naturally leads to false positives. These false positives will later be mitigated by our model as will be explained in Chapter 6.

Program Construct	**Remopla Representation**

`if(cond){`
`...`
`}else{`
`...`
`}`
	`if`
	`::  true -> ...;`
	`::  else -> ...;`
	`fi;`

`f(){`
`...`
`}`
	`module void f(){`
	`move_state(ACTION_-`
	`FUNCTION_CALL_f);`
	`...`
	`move_state(ACTION_-`
	`FUNCTION_RETURN_f);`
	`}`

`f();`	`f();`
`f(`v_0`,...,`v_n`)`	`ARG[0]=ARG_`v_0`;...;ARG[n]=ARG_`v_n`;f();`

Table 4: Remopla Representation of Program Constructs

As shown in the second entry, a function is represented as a Remopla module of type `void` and without parameters. Two important program actions are associated with each function: the entry of the function and its exit. We express these actions explicitly and embed them in the program model. These actions are passed as parameters to the `move_-state` module, which in turn checks them against the defined transitions and changes the automaton state accordingly.

```
#include <stdio.h>
#include <stdlib.h>
...
void encrypt(char * buffer, char *key);
void save(FILE *outfile, char *buffer);

int main (int argc, char *argv[])
{
   ...
1   printf("Key required; please enter the value of the
   encryption key: \n");
2   int keyval;
3   scanf("%d", &keyval);
4   if(keyval > 9999){
5      if(keyval > 999999){
6         printf("Invalid key.\n")
      }else{
7         char *key = (char *)malloc(MAX_LEN);
8         memset(key, '\0', MAX_LEN);
9         snprintf(key, MAX_LEN, "%d", keyval);
10        encrypt(buffer, key);
      }
    }else{
11     printf("Invalid key.\n");
12     exit(1);
    }
13  save(outfile, buffer);
14  printf("File %s has been encrypted & saved.\n", argv[1]);
    ...
15  return 0;
}
```

Figure 23: Sample Code for Illustration of Remopla Model Generation

The last two entries defines respectively the mapping for function calls with and without parameters. Before a function is called, its arguments are stored in the global Remopla array `ARG[]`, indexed by the parameter's position in the function's signature. When the property module is checking a given program action, this global array is inquired for the passed parameters. For example, Figure 24 shows the Remopla program for the sample code in Figure 23. For clarity, Figure 24 shows only statements relevant to our discussion. The variables `actions` and `args` contain all the program actions and the passed parameters, respectively. The initial state of the program corresponds to the initial state of the considered security automaton. The program model can be in one of the states defined in the given property automaton (i.e. the enumeration variable `states` in Figure 22). The state of the model (i.e. `current_state` in Figure 24) is synchronized with the property automaton (i.e. the one in Figure 22) by invoking the `move_state()` module with the current program action as a parameter. In this example, line 32 in Figure 24 is reported by our tool as a violation of the property in Figure 22.

Suspicious Paths Detection and Reporting SVR process concludes by detecting program paths that could potentially violate the security property in concern. While SVR is one of the components composing our model, it should be noted that SVR can be executed as an independent security violation detection tool. However, due to the nature of static analysis, the detected vulnerabilities reported by SVR may very well include false positives. While this may be acceptable in some cases, in other cases it may be massive and overwhelmingly unacceptable. Consequently, the decision of whether to run SVR independently is left to the security analyst. Since our goal is to eliminate any reporting of false positives, we do refer to the detected paths as suspicious paths. We then inject those paths to the other components of our model, where dynamic analysis is conducted to verify the real existence of such reported vulnerabilities, as we explain in Chapter 6.

```
//Automata actions declaration:
enum actions {ACTION_FUNCTION_CALL_encrypt,
ACTION_FUNCTION_CALL_save,
          ACTION_PROGRAM_END};
//Function call possible arguments:
enum args {ARG_buffer,ARG_key,ARG_outfile};
//ARG array declarartion:
int ARG[10];
int current_state;

module void encrypt (){
move_state(ACTION_FUNCTION_CALL_encrypt);
return;
}
module void save (){
 move_state(ACTION_FUNCTION_CALL_save);
 return;
}
module void main (){
 if       //if(keyval > 9999)
 :: true ->
   if    //if(keyval > 999999)
   :: true -> skip;
   :: else ->
      ARG[0] = ARG_buffer;
      ARG[1] = ARG_key;
      encrypt();
   fi;
 :: else -> break;
 fi;

 ARG[0] = ARG_outfile;
 ARG[1] = ARG_buffer;
 save();

 return;
}
```

Figure 24: Remopla Model of Code in Figure 23

5.4 Moped-Based Test-Data Generation for Reachability Analysis

As we have previously indicated in Chapter 2, various test-data generation approaches exist. Such approaches include random test-data generation [18], directed random test data generation [63], path-oriented test data generation [20, 28, 34], genetic and evolutionary algorithms [30, 32], and goal-oriented [92] such as the chaining approach [54]. These approaches vary in nature and target different goals.

If vulnerabilities are present in a software, then they are present at specific program points, which we refer to as *security targets*. Test-data generated by random testing may never lead to these points. Full-path coverage may result into a massive effort being wasted in exploring paths in the software that are not at all related to the vulnerability targets in question. A goal-oriented approach may succeed in generating data to reach the goal, but through an irrelevant path; in other words, a path that is not vulnerable. Consequently, to achieve our security detection goals, we had to design another test-data generation model, which takes advantage of a reachability checker, referred to as Moped checker. The following subsections provide the details of our reachability test-data generation component.

5.4.1 Reachability Checker

The reachability checker used in our model, namely Moped, is a model-checker for push-down systems [88]. The checker simulates program execution for all possible arguments within a finite range of values and generates trace information of these executions. Given a target in a Remopla representation, the checker attempts to verify whether 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. Consequently, there is a large performance overhead for these generations to be made. Generally, there are two major factors that affect the performance of the model-checker: the value ranges and the number of the variables.

The checker as designed is not suitable for our purpose of security test-data generation since its concern is to produce data to reach a specific target without any regard to a specific path. For our system, we require the data to be generated for a specific path; in other words, reaching the target through any path is insufficient. Consequently, we needed to modify the way the checker works so that the checker is forced to either generate data for a certain path, or to fail to generate data for the path, on which we consider as infeasible in such case. Additionally, we need to enhance the performance of the checker. To achieve these goals, we revert to program slicing to optimize the Remopla representation. Section 5.4.2 details the program slicing component of our model.

5.4.2 TDG through Moped Checker and Program Slicing

Program Slicing has been introduced in [159] as a method of abstracting programs and reducing them to a minimal form that is still capable of producing an original subset of a behavior. A slice 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.

Figure 25 shows the process of generating appropriate vulnerability reporting using Moped checker and program slicing. The program is first passed to SVR, which reports a set of suspicious paths that could violate the property in concern. These paths along with the source code are then passed to a static analyzer, which performs data and control flow analyses to determine what is required for these paths, such as the set of relevant variables. Based on these analyses, program slicing is performed and the sliced program is finally passed to the Moped Checker for test data generation, and vulnerability reporting.

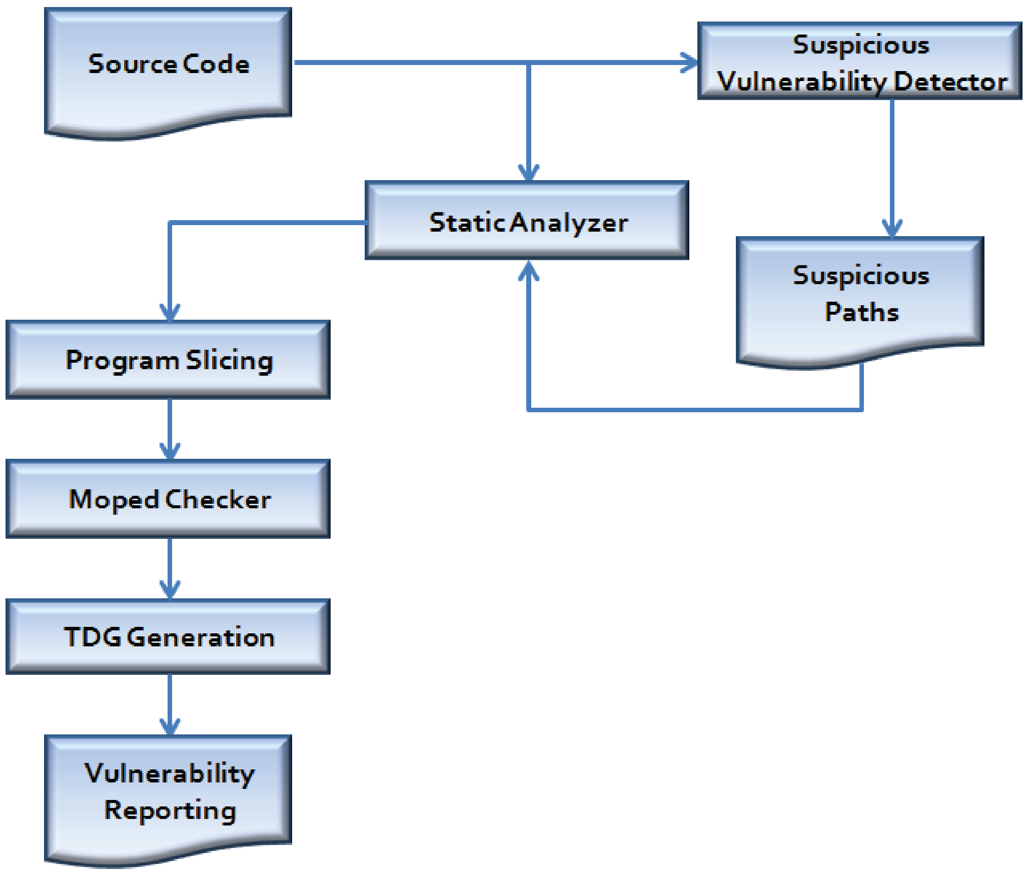


Figure 25: TDG using Moped Checker and Program Slicing

Figure 26 and Figure 27 provide an illustrative example of the process, where the software is to be tested against violations of security property involving `vfork()` and `execcve()` system calls. Subsequent calls to `vfork()` that precede the call to `execcve()` have the potential vulnerability of modifying the state of the parent process instead of the child process. As shown in Figure 27, the slicing window includes the suspicious path as well as variables `x1` and `x2`, which are the only relevant variables to that path.

```
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(){}
```

Figure 26: Sample Remopla Representation without Slicing

5.4.3 Reachability Analysis Case Study

We conducted some experiments on our developed system. All the experiments are performed under the following environments: Operating System: Ubuntu Linux release 6.10, Linux Kernel: Linux kernel version 2.6.17-11-generic, Java Runtime Environment: Java(TM) SE Runtime Environment (build 1.6.0 02b05), Shell: GNU bash version 3.1.17(1)-release, autoconf: GNU Autoconf version 2.60, make: GNU Make 3.81, and Moped version 2.

```
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(){}
```

Figure 27: Sliced Remopla Representation of Code in Figure 26

These experiments target various aspects including the detection power of the system, the performance of the test-data generator component, and the scalability of the analysis. Our system is verified against various software, including gzip verion 1.2.4, httpd version 2.2.8, Openca-tools version 1.1.0, OpenSSH version 5.0p1, Shadow version 4.1.2.2, and Sudo version 1.7.0. Concerning scalability and detection power, our tool successfully verifies more than 20 reachability properties over the tested software.

For measuring the performance enhancement using our slicer for test-data generation, we conduct various experiments over small software, as well as over larger-size software; in particular gzip verion 1.2.4 (96 files totalling to 24247 lines of code). Figure 28 shows the results of these tests. In general, our model provides a large performance enhancement, of up to 99.8%, especially when the value range of the variables is increased. In addition to this experiment, further experiments are conducted on the system for both validation and comparison reasons. The details of such experiments are deferred to subsection 7.6.3 in

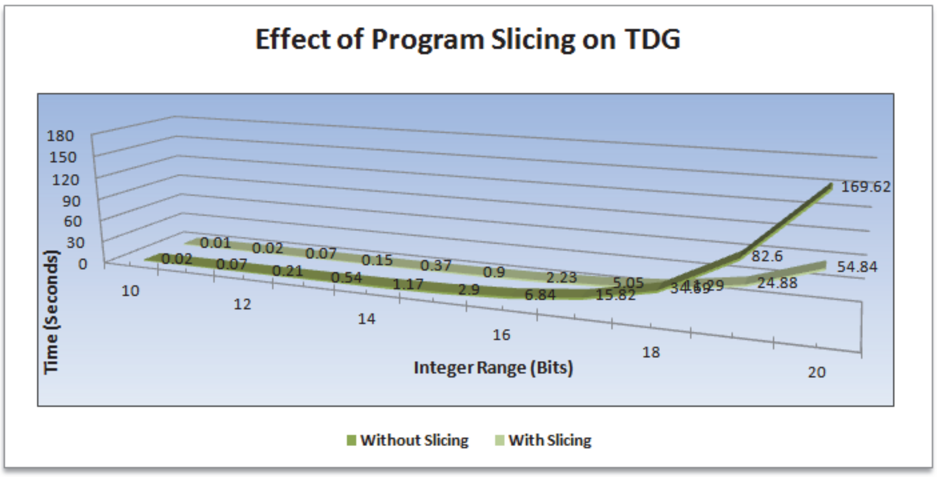


Figure 28: Slicing's Influence on Test-Data Generation

5.5 Examination of Static-Analysis Based Approaches

As described in Section 5.3 and Section 5.4, both SVR and Moped-based test-data generation approaches are heavily based on static analysis. Consequently. they are limited to the detection powers of such analysis. For SVR, the detection of vulnerable paths is reported by statically investigating the paths of the source code against potential vulnerabilities. Hence, and due to the nature of the static analysis, false-positive reports are very possible. Depending on different factors, such as the size and the complexity, of the software being tested, the amount of false-positive reports may be significantly unacceptable. To mitigate the problem, some sort of dynamic analysis is needed so that all reports are guaranteed to reflect real vulnerabilities.

Furthermore, Moped-based test-data generation is restricted by the static-analysis nature of the model-checker as well as other Moped limitations. In particular, the checker is capable so far of handling only integer and boolean types. The integer range for data generation is also limited, which adds another restriction.

5.6 Conclusion

We have presented in this chapter our approach to reachability analysis. In this context, we have presented our classification to security properties as being either reachability or non-reachability ones. We have then explained how some non-reachability properties can be reduced to reachability ones. There are two major issues that are related to this subject, which are the detection of potential vulnerabilities in a software and the generation of test-data to prove that such vulnerabilities can indeed be exploited at runtime. The latter is needed to eliminate false-positive reporting.

As part of this research effort and contribution, we have implemented a framework that targets the detection of vulnerabilities and the generation of test-data to prove their existence. Two major components compose our framework. First, a vulnerability detector, namely, Static Vulnerability Revealer, and second, a test-data generator based on Moped model checker. Case studies have been conducted to validate our approaches.

We have further analyzed the framework and its components against limitations caused by their dependence on static analysis. Due to these various limitations, we conclude that such tools, which are heavily based on static analysis, can be very useful and more suitable for testing relatively small software, such as embedded software. For testing large systems, we need to have other approaches and tools that combine the powerful capabilities of both static analysis and dynamic analysis. We refer to such approaches as hybrid analysis approaches. This represents the subject of our next contribution in this research, which will be detailed in Chapter 6.

Chapter 6

Hybrid Approach to Test-Data Generation

6.1 Introduction

The main intent of this chapter is to ascribe a novel approach to test-data generation that takes advantage of the combined powerful capabilities of both static and dynamic analyses. The research reported in this Chapter, leverages some of the ideas that emerged from our fruitful collaboration with H. Z. Ling [69], that were published in [9, 70, 71]. Nevertheless, we extend them by adding more rigor components and approaches when it comes to the test data generation for linear and non-linear constraints. Indeed, we provide a detailed investigation of the needed algorithms to solve the linear/non-linear collected constraints. For instance, we propose the idea of root-crossing minimization. Moreover, we explore the use of the Broyden method [24] for multi-dimensional root finding. Our contributions in this domain can be summarized as follows:

- Elaborating a novel approach to test-data generation, namely hybrid static-dynamic test-data generation approach.

- Elaborating a general framework for automating security testing, including security vulnerability detection and test-data generation.

As discussed in Chapter 2, various test-data generation techniques exist. However, these techniques are heavily based on either static analysis or dynamic analysis. Consequently, such techniques suffer limitations caused by the utilized approach. Additionally, many of these techniques may be capable of generating test-data for general testing purposes, which makes them inappropriate for security testing in particular. For instance, techniques that are capable of generating test-data to reach a particular program point may indeed generate such data. Yet the execution with the generated data may lead the program to traverse a non-vulnerable path, disallowing any vulnerability detection. Techniques that use full-path coverage have the potential of producing an overwhelmingly unacceptable number of generation attempts, where a lot of effort may be wasted on traversing irrelevant paths to the potentially vulnerable ones.

In Chapter 5, we have examined the utilization of static analysis. The obtained results concluded that such approaches suffer some limitations and result in the generation of false-positives. In order to overcome such deficiencies, we resort to dynamic analysis. More precisely, we use a synergy between the two types of analyses. In effect, as part of this research, we examine two major concerns: 1) What type of analysis is appropriate for security testing purposes?, and 2) What is the needed approach for generating test-data for the purpose of security vulnerability detection?

The remainder of this chapter details the results of such an effort. In Section 6.2, we introduce our approach to test-data generation, namely Goal-Path-oriented System (GPS). Section 6.3 details our dynamic approach to test-data generation, which considers both linear and non-linear programming. In that context, we also introduce a novel approach to the subject, namely Root Crossing Minimization (RCM). Section 6.4 introduces our hybrid framework for automating vulnerability detection, and provides the details of the main components composing this system. Finally in Section 6.5, we provide a summary and conclusion of the subjects covered in this chapter. We defer the details of the experiments conducted over our system to Chapter 7.

6.2 Goal-Path-Oriented System - GPS

Various approaches exist for data-generation as described in Chapter 2. These approaches can be classified based on their scope of coverage. For instance, random generation attempts to repeatedly generate data that may eventually satisfy the purpose of the generation. This approach however does not guarantee that such data can ever be produced in a bounded time, neither it guarantees that the generated data would cover all possible paths of the software being tested. Directed-random generation approaches attempt to reduce some of the shortcomings of random generation. While the data is still randomly generated, it would drive program execution through particular paths. In contrast to random generation, approaches based on full-path coverage would attempt to generate at least one set of data that would allow the traversal of each path in the program. Goal-oriented approaches, such as the chaining approach, have rather a significantly different focus where neither random generation of data is performed, nor full-path coverage is required. Instead, such approaches attempt to generate data with which a specific program point is reached.

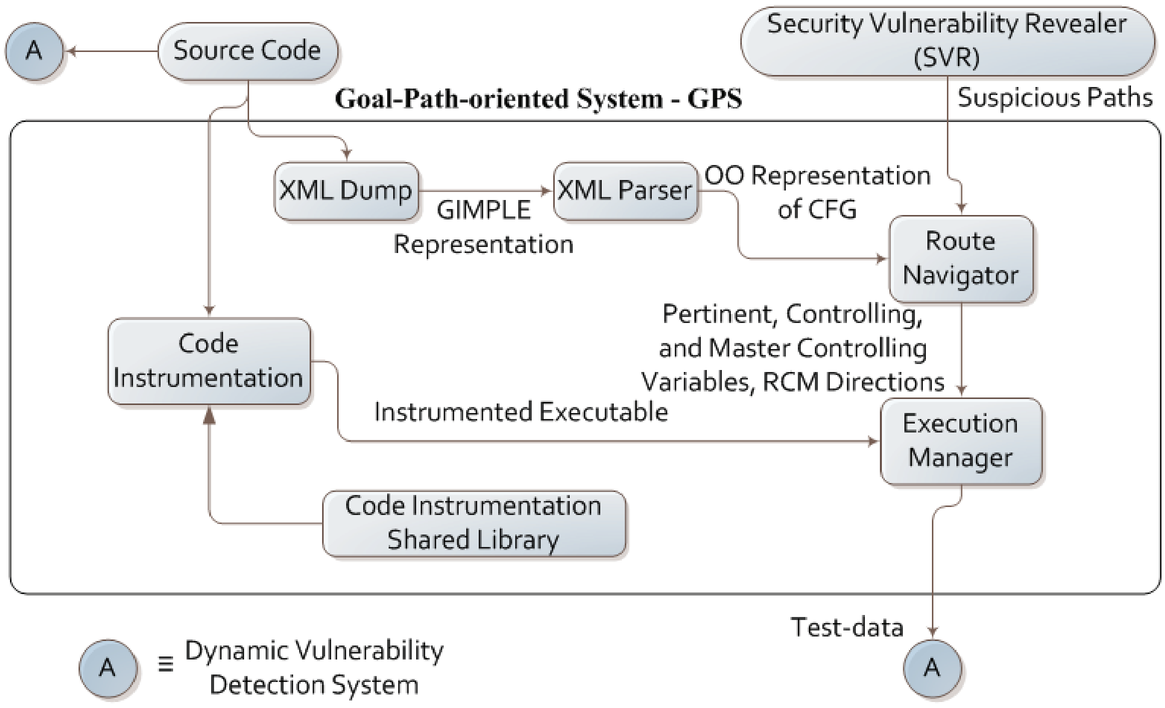
For the purpose of security vulnerability detection, none of these approaches is appropriate. We hence need to view the problem at hand differently. Our definition of the problem is stated as follows: "Given a target point *t* in a program, and a path p to reach that target, find program inputs $x_1, x_2, \ldots, x_n$, with which *t* can be reached through the execution of *p*". Our view of the problem is driven by the fact that a specific security vulnerability violating security property presents itself at a specific program point. Yet, merely reaching this point through any execution path may not result in the violation of the security property. We consequently define a new approach to test-data generation, namely the Goal-Path-oriented System (GPS). An overview of GPS is shown in Figure 29. Below, we provide the details of the major components composing our system.

6.2.1 Route/Path Navigator

Before proceeding to the details of the components, the following definitions in relation to a program path are given:

Basic Block: A sequence of contiguous instructions that contain no jumps or labels. Hence, a basic block has only one entry point and one exit point. If the first instruction in a basic block is executed, all instructions of the block will then be executed as well.

Conditional Basic Block: A basic block that ends with a conditional statement. A conditional statement has the following format: `lhs` *`op`* `rhs`, where the `lhs` and `rhs`



A
Source Code
Security Vulnerability Revealer (SVR)
Goal-Path-oriented System - GPS
Suspicious Paths
XML Dump
GIMPLE Representation
XML Parser
OO Representation of CFG
Route Navigator
Pertinent, Controlling, and Master Controlling Variables, RCM Directions
Code Instrumentation
Instrumented Executable
Execution Manager
Code Instrumentation Shared Library
Test-data
A
A ≡ Dynamic Vulnerability Detection System

Figure 29: GPS Architecture

respectively represent the operands on the left-hand and right-hand sides of a comparison operator op. Generally, op is one of the following operators: $==, !=, >, \geq, <, \leq$.

Execution Path: A sequence of basic blocks.

Suspicious Path: An execution path in a program that has the potential of violating a security property.

Critical Branch: A program branch that would permanently drive execution away from a suspicious path.

Problem Node: A conditional basic block where the execution of the program leads to a critical branch.

Required Branch: A branch of a suspicious path that is initiated at a conditional statement.

Controlling Variable: A variable appearing in a conditional statement along a suspicious path.

Pertinent Variable: A variable appearing along a suspicious path in the program.

Master Controlling Variable: A pertinent variable that is influential to controlling variables. Data generation is needed for such variables.

Constraint Function: A function defining some prescribed conditions. The format of such conditions depends on the truth value of a conditional statement, where the execution of a particular branch is to take place.

Root Direction: The execution direction towards finding the root of a constraint function. The root, x, of a function f is defined such as $f(x) = 0$, where x is a member of the domain of f.

Root Crossing Minimization (RCM): The process of finding a member, x, from the domain of a function f, such that $f(x) < 0$.

Constraint Value: This is a concrete generated value achieving root crossing minimization of a constraint function. Such value is required for dynamic execution of the program.

We tend to view the different paths of a software execution as routes of a city. Applying this conceptual view, the problem of finding test-data that allow a specific route to be taken can be looked at as finding all relevant conjunctions along a route and determining which route to be taken at each of these conjunctions. Following the same analogy, we also understand that there may be controlling traffic signals or gates along the route. Such controlling entities do in fact determine/control the flow to the destination. Additionally, the existence of a massive number of traffic lights and gates within the city is possible. While this may indeed be the case, only a bounded number of these controlling entities may be pertinent/relevant to the specific route to be taken. Hence, given a suspicious path, the goals of the path navigator component of our model can be described as follows:

- Determine the exact set of *pertinent variables* to be generated for a given path;
- Determine the exact set of *controlling variables* along the path;
- Determine, for every conditional statement, the exact set of variables that are influential to the controlling variables. This set hence includes the *master controlling variables* of that path. The generation of data actually takes place for those variables;
- Determine the *RCM directions* to be taken at the different conjunctions along the path, so that the execution of a required path is achieved.

Consequently, further static processing is needed for the path navigator to achieve the needed operations. Particularly, both static data-flow and control-flow analyses need to be performed.

Our static analysis is not performed directly over the source code; rather over a GCC GIMPLE representation of it. As shown in Figure 29, we first perform a transformation of GIMPLE into XML, using a GCC patch [51] that creates an XML dump. The resulting

XML files are parsed to produce an object-oriented class representation of GIMPLE. The path navigator then performs its static analysis over the data-flow and control-flow object-oriented representation of GIMPLE. Figure 30 shows a partial GIMPLE class diagram representation.

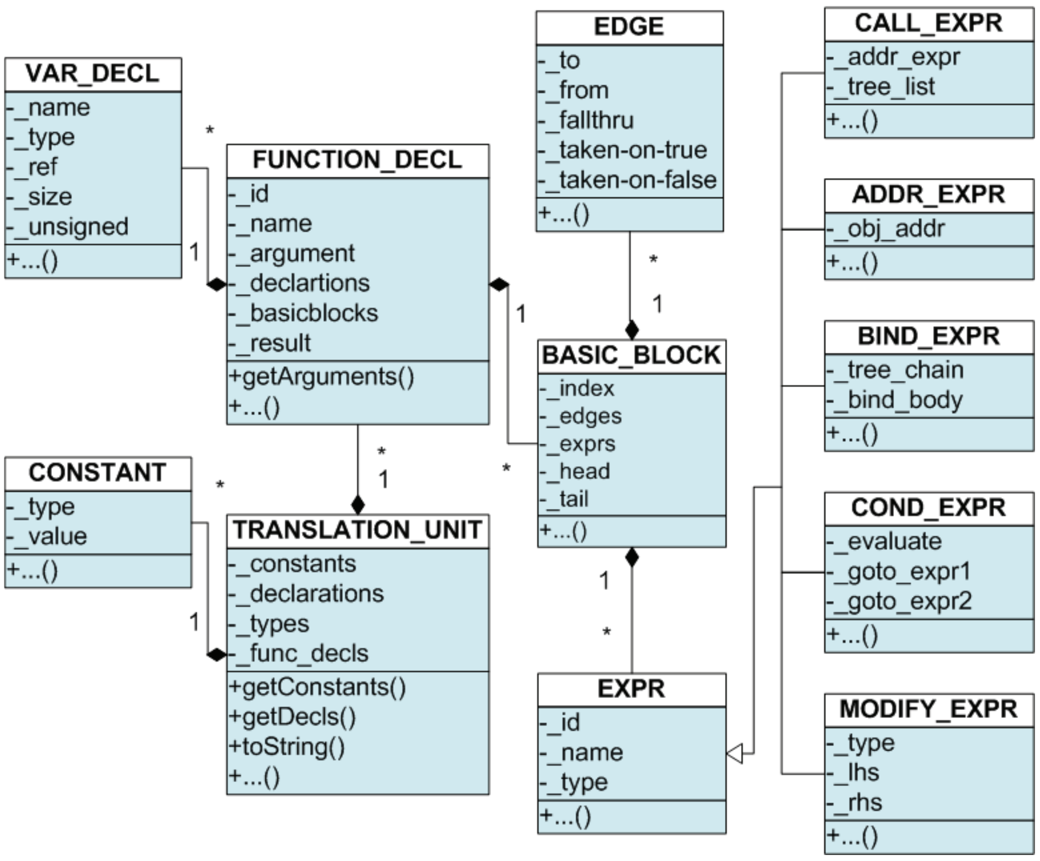


Figure 30: Partial GIMPLE Class Diagram

Sections 6.2.2 and 6.2.3 highlight the static analysis phases performed by the path navigator.

6.2.2 Static Data-Flow Analysis Phase

In this phase, we perform a set of static analysis, which is particularly needed in order to support data-flow analysis. This analysis takes place over the GIMPLE representation of the program. The purpose of this analysis is to trace the particular variables that are related to the suspicious path that is under investigation. Specifically, we trace the following:

- Pertinent variables to the suspicious path. This allows us to trace all program variables, including input variables, that may affect the execution of the suspicious path being investigated. This analysis consequently filters out all other program variables that are irrelevant to that path. Practically, this could be a very large set of variables;
- Controlling variables at all nodes along the suspicious path;
- Relations between pertinent variables, including input variables and controlling variables. With this, it is possible to trace all the variables actually controlling the execution of the path, which we refer to as master controlling variables. Data generation is needed for these variables.

Tracing the dependency between pertinent variables and controlling variables is crucial for our test-data generation. While controlling variables seem to be the ones controlling the nodes of the path, and hence data generations attempts are needed over them to enforce path execution, this may very well not be the case in reality. The reason behind that lies on the dependency between these variables and other ones, especially input variables. To briefly illustrate, assume a node along the suspicious path is controlled by the following statement "`if (x > 1000)`", where the truth value of the statement must evaluate to `false` for the execution to follow the required path (i.e. execution of the `else` branch of this `if` statement). Further, assume a previous statement leading to this node is as follows: "`x = 2 * y + 20;`", where `y` is an input variable. Attempts to generate values for `x` would rather be inappropriate since `x` is actually influenced by `y`. Hence the generations are actually needed over `y` to force the execution to follow the required path. Our static data-flow analysis traces all these types of relations, allowing us to conclude the exact variables that are needed for the generation process.

Beside performing the needed functionalities, the analysis results in performance enhancement. Since a great set of program variables may be filtered out as irrelevant by such analysis, no generation will ever be attempted for variables in that set. In a reasonably large-size software, eliminating generation attempts over such irrelevant variables may have a significant impact on performance, compared to other approaches such as random test-data generation for instance.

6.2.3 Static Control-Flow Analysis Phase

The purpose of this analysis is to trace and classify all branches out of conditional nodes along the suspicious path being investigated. This analysis is actually needed in support to the dynamic analysis phase as we explain in Section 6.3. Upon the generation of concrete variables for the master controlling variables, program execution is needed to validate whether the generated values actually lead the execution as desired. In general, there are no guarantees that the generated values would lead to the proper execution. Such values may actually result in the execution being permanently driven away from the path. In this phase, all conditional branches at the nodes of the path are traced and qualified as either *required*, or *critical*. A critical classification is given to paths permanently leading the execution away from the required path. Dynamic examination of these classifications takes place at execution time.

6.3 Dynamic-Based Approach to Test-Data Generation

One of the major disadvantages of static-based approaches is the potential generation of false-positives. In many cases, especially when testing relatively large-size software, this can be massive and possibly beyond being acceptable. To eliminate such invalid reports, dynamic analysis is needed. In particular, concrete data is to first be generated then program execution is to be conducted with such data to prove the actual existence of the vulnerability.

In this phase, our approach conducts a set of dynamic analysis to fulfill such requirements. In general, the analysis performs various runtime computations and history tracking for the purpose of generating concrete data, with which the vulnerability is executed. Various components are needed to achieve our goals, including different code instrumentations and runtime monitoring. Sections 6.3.1, 6.3.2 and 6.3.3 detail the different components composing our dynamic test-data generation model.

6.3.1 Code Instrumentation for Dynamic Monitoring

Various code instrumentations are needed for the purpose of dynamic monitoring of the program as well as for the automation of test-data generation. The instrumentations conducted by our model to achieve such goals are as follows:

Input Abstraction Instrumentation: Software programs are composed of two phases, execution phase and Input/Ouput (I/O) phase. The I/O phase is needed to provide the program with the needed data required for the execution to take place. These data can be obtained from different sources, such as input files, database tables, interactive user-input, etc. Since our model targets the automation of the test-data process, user interaction is to be elimenated/minimized. Input abstraction code instrumentation is needed for that purpose.

With this instrumentation, the original program is modified so that all user-interactive inputs are suppressed. In particular, these interactive calls are substituted with various calls that allows the data to be provided without any user interaction. For instance, calls to methods such as `scanf()` are first suppressed by this instrumentation, and replaced by other calls that automatically supply the proper needed input without user interaction. Currently, our input abstraction instrumentation is capable of substituting a specific set of known C/C++ input calls. Further extensions are however possible without much difficulties.

History-tracking Instrumentation: Data generation is needed for the purpose of allowing the execution to follow the desired vulnerable path. However, there are no guarantees that the generated data would lead to such execution. In such cases, generated data may actually lead the execution permanently away from the desired path. While the generated data in such cases is not satisfactory, this improper data may yet represent very useful information that can drive the following generation attempts. This type of instrumentation is needed for our model to keep track of such previously generated improper data. The model uses such data to determine the following generations.

Execution Monitoring Instrumentation: This instrumentation injects monitoring code that controls program executions. Upon the generation of test-data, attempts are made to execute the suspicious path with such data. Should the data drive the execution to hit a critical path, the current execution is halted. Additionally, the collected information from the failed attempt is exchanged with the data-generation server as explained in Section 6.3.3. This instrumentation is needed for that purpose.

6.3.2 Dynamic Test-Data Generation

Our static analysis phase reveals the needed information to track a suspicious path, including problem nodes and controlling variables. It is then desired to generate data that would drive the execution through this path. To achieve that, our model converts all conditional statements at problem nodes along the path to constraint functions. In fact, the constraints functions are based on the master controlling variables. Further, the constraints are defined in relation to a value of 0. The definition of a constraint function is hence given as follows: $C(x)\ \{<, \leq, =, \neq\}\ 0$, where *x* is the set of master controlling variables along the suspicious path. Given a general conditional statement, such as `if(lhs op rhs)`, at a given problem node along the path, Table 5 illustrates how constraint functions are constructed and what constraint values are needed accordingly. lhs and rhs represent the left-hand and right-hand sides of the comparison statement accordingly, where *op* is one of the comparison operators.

Conditional Statement	**Required Branch**	**Constraint Function**	**Desired 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)	$\neq 0$
lhs $\neq$ rhs	False branch	abs(lhs $-$ rhs)	$= 0$

Table 5: Constraint Functions

With that in mind, the test-data generation problem is then converted to a variation of the root finding problem. We refer to this variation as Root Crossing Minimization (RCM). The generated data is to allow the constraint function at the nodes along the suspicious path to first reach the root, if necessary, then passes beyond it, as we explain shortly. Additionally, since this operation is performed dynamically at runtime, where time overhead is critical, such convergence to the root must be done as quickly as possible. Our model analyzes the constraint function at early generation steps to judge whether it is linear or non-linear. Based on that, the model determines the appropriate data generation technique to be used. Below, we detail these techniques.

Testing for Linearity

The constraint function for which data generation is needed may either be linear or non-linear. It is important for our model to know such details since this determines the test-data generation process. Additionally, it is important to point that our static analysis phase traces variable dependency and is capable of determining the actual controlling variables. However, the analysis neither trances, nor attempt to construct, the actual form of these constraint functions. In other words, the exact symbolic form of the function and how the variables relate to each other is unknown to us. Consequently, all needed operations over these constraint functions must be performed numerically and not symbolically.

To detect the linearity a constraint function, *f(x)*, our model starts by constructing two sets of uniformly dispersed and independently selected values $\{x_0, x_1, \ldots, x_{n-1}\}$ and $\{y_0, y_1, \ldots, y_{n-1}\}$, for some bounded integer $n > 0$. The model then repeatedly execute the function twice for each x_i, and y_i, where $i = 0..n-1$. The constraint function is assumed to be linear if:

$$f(x_i) + f(y_i) = f(x_i + y_i),$$

for all attempted x_i and y_i values. A failure for this rule to hold true for all attempts is indicatives of the non-linearity of the function.

Linear Test-data Generation

Our model starts the process of data generation for each constraint function by detecting whether or not the function is linear. Should this process concludes that such relation is linear, the model utilizes a one-dimensional search procedure to generate the data. As an initial step, the model starts by conducting an exploratory search to determine the direction of data generation in relation to a starting random point. The search initiates a random value in a hope that such value would represent an appropriate constraint value. If this is not the case, the model utilizes a dichotomy technique by applying a small change to the initial value, and attempts again. The purpose of this small modification is to determine whether or not the new data is heading the constraint function towards root crossing minimization. If this turns not to be the case, a small difference in the opposite direction to the initial value is attempted. If multiple variables compose the controlling variables set, the generation is applied to one of them at a time, while keeping the rest constant.

Following the exploratory step, a pattern search is conducted. The goal of this search is to achieve root crossing minimization as quickly as possible. For that, a large offset is applied to the last generated data, then program execution is initiated. If the data fails to achieve the required minimization, yet it heads the constraint function towards being minimized, then another large offset is applied following the same generation direction. Otherwise, a large offset is applied to the opposite direction. A general description of our linear data generation is given in Algorithm 1.

Non-linear Test-data Generation

If the linearity test of the function reveals that it is non-linear, our model initiates a different test-data generation component designed for non-linear functions. Below, we first provide a brief relevant background to the subject of non-linear programming, then follow with the details of our test-data generation component.

Algorithm 1 Test-data Generation Algorithm for Linear Programming

```
$Input$: Suspicious path, instrumented program for TDG,
$Output$: Generated test-data

$x \Leftarrow initial\_guess\_value;$
**while** unable to pass problem node && attempt limits are not exhausted **do**
   // Execute program with $x$, and track root crossing minimization
   $c(x) = execute(x);$
   **if** $suspicious\_path$ is executed **then**
      // Minimization already achieved
      $return\ x;$
   **end if**
   /*Adjust data with a small offset, $\varepsilon$, in one direction,
   and re-execute program with the new value*/
   $x \Leftarrow x + \varepsilon;$
   $y_1 \Leftarrow x;$
   $c(x) = execute(x);$
   **if** $suspicious\_path$ is executed **then**
      // Minimization already achieved
      $return\ x;$
   **end if**
   /*Adjust data with a small offset, $\varepsilon$, in opposite direction, and re-execute program*/
   $x \Leftarrow x - \varepsilon;$
   $y_2 \Leftarrow x;$
   $c(x) = execute(x);$
   **if** $suspicious\_path$ is executed **then**
      // Minimization already achieved
      $return\ x;$
   **end if**

   /*If this point is reached, pattern search is to be initiated*/
   /*Determine direction of exploratory search based on
   history tracked information from exploratory step*/

   $expolartory\_direction\_op =$
          $determine\_direction\_operator(y_1, y_2, c(y_1), c(y_2));$
   /*Determine appropriate previous exploratory value based on
   tracked information from exploratory step*/

   $x \Leftarrow determine\_exploratory\_value(y_1, y_2, c(y_1), c(y_2));$
   // Initiate pattern search by applying a large offset, $\Delta$, to previous $exploratory\_value$,
   and re-execute
   $x \Leftarrow x$ expolartory_direction_op $\Delta$;
**end while**
```

Newton-Raphson Optimization

Newton-Raphson [114] is an iterative method for finding the root of an equation. Generally, given a differentiable function *f(x)*, the method can be used to find the stationary points of *f(x)*. The method starts with an initial guess, x_0, with the purpose of finally converging to some stationary point x_*, such as $f'(x_*) = 0$, where $f'(x)$ is the derivative of $f(x)$. The iterative equation of Newton-Raphson is defined as follows:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)},$$

where $i >= 0$.

The method has many advantages. If there is a solution, the method has a fast convergence rate, which is quadratic. Additionally, the method requires only one initial guess. On the other hand, the method has some drawbacks, including the possibility of division by zero if the value of the denominator evaluates to zero. Other problems with the method include the problem of root jumping [124]. Depending on the form of the method, if the initial guess is close enough to the root, then the method may fail to find that closet root.

While the above drawbacks may indeed represent problems, there is one particular drawback of the method that is critically significant for our model. For Newton-Raphson to be applied, the derivative of the function, $f'(x)$ must be symbolically known. As we previously discussed, our static analysis does not trace or attempt to construct the symbolic representations of the constraint functions. Consequently, it is not feasible to calculate $f'(x)$ symbolically, and hence it is not possible to use Newton-Raphson.

The Secant Method

The Secant method is a root-finding algorithm that uses approximation techniques to find the root of a method. If the derivative of the function, $f'(x)$, cannot be symbolically calculated, it can alternatively be approximated. Using Backward Divided Difference scheme [123], the Secant method approximates $f'(x)$ as follows:

$$f'(x_i) \simeq \frac{f(x_i)-f(x_{i-1})}{x_i-x_{i-1}},$$

where $i >= 1$.

With that, there is no longer a need to find the exact symbolic formula of $f'(x)$. However, the method requires two initial guesses in contrast to Newton-Raphson, where only one initial guess ia required. Now, considering the approximation of $f'(x)$, the iterative equation of Newton-Raphson can be stated as follows:

$$x_{i+1} = x_i - \frac{f(x_i)}{\frac{f(x_i)-f(x_{i-1})}{x_i-x_{i-1}}} = x_i - \frac{f(x_i)(x_i-x_{i-1})}{f(x_i)-f(x_{i-1})},$$

where $i >= 1$. This actually defines the Secant method formula for finding the root of $f(x)$. As stated, the method uses two initial guesses, x_{i-1} and x_i to find x_{i+1}. The method then uses x_i and x_{i+1} to find x_{i+2}, and so on, until a root is found. It should be noted that the initial guesses may not bracket the root, and so the Secant method is an *open* method. Open methods for finding functions roots are not restricted to intervals. Such methods usually have quick convergence, but they do not guarantee finding the root, even if one exists. Methods that use bounded intervals that bracket the root are referred to as *closed* methods. Such methods usually converge slowly but they always find the root if it exists [110].

Multi-Dimensional Root Finding - The Broyden's Method

The Secant method is suitable for root finding where the function is only one-dimensional. Broyden's method [24] is a generalization of the Secant method for multi-dimensional functions. The method considers the solving of a set of non-linear functions in contrast to only one function as with the Secant method. Consider the vector $\vec{F}$ is a function of the vector $\vec{x}$ such that:

$$\vec{x} = (x_1, x_2, \ldots, x_k), \text{ and}$$
$$\vec{F}(\vec{x}) = (F_1(x_1, x_2, \ldots, x_k), F_2(x_1, x_2, \ldots, x_k), \ldots, F_k(x_1, x_2, \ldots, x_k)),$$

where k is the number of functions to be solved. The method attempts to solve the following equation:

$$\vec{F}(\vec{x}) = \vec{0}.$$

Now generalizing the Secant method formula accordingly, Broyden's method formula can be stated as:

$$\vec{F'}(\vec{x_i}) \simeq \tfrac{\vec{F}(\vec{x_i}) - \vec{F}(\vec{x}_{i-1})}{\vec{x_i} - \vec{x}_{i-1}},$$

where $i >= 1$. The method then replaces the derivative $\vec{F'}$ with the Jacobian matrix, J, which is an n-by-m matrix defined as:

$$J = \begin{bmatrix} \frac{\delta F_1}{\delta x_1} & \cdots & \frac{\delta F_1}{\delta x_n} \\ \vdots & \ddots & \vdots \\ \frac{\delta F_m}{\delta x_1} & \cdots & \frac{\delta F_m}{\delta x_n} \end{bmatrix}$$

The Jacobian matrix hence includes all first-order partial derivatives of vector function with respect to another vector. Broyden's method determines the Jacobian matrix iteratively using the Secant equation. Effectively, the Broyden's formula is stated as:

$$J_i \simeq \frac{\vec{F}(\vec{x_i}) - \vec{F}(\vec{x}_{i-1})}{\vec{x_i} - \vec{x}_{i-1}},$$

where $i >= 1$ is the index of the iteration.

Root Crossing Minimization (RCM)

As discussed, Newton-Raphson, the Secant method and Broyden's method, all attempt to find the root of a non-linear equation. Other optimization techniques, such as Quasi-Newton [114], attempt to minimize the function to find the minima of such function. For our test-case generation, none of these techniques is suitable since we are neither interested in finding the root, nor in the minima of the method. In specific, we are exactly interested in finding a value with which the function would cross below the threshold of zero. Such value is sufficient for our model to pass a problem node, so the traversal of a suspicious path is successful. It is true that if a minima exists below zero, then finding such a value would automatically satisfy our condition. However, such minimization techniques may result in a lot of extra unnecessary effort if a value below zero is found, but yet it is not the minima searched for by the algorithm. Root finding algorithms on the other hand suffer two problems in relation to our test-data generation concerns. Firstly, finding the root is only sufficient in the cases where the constraint function is testing for equality. For the more general case, where a value that crosses the threshold of zero is needed, the algorithm stops short from generating the needed value. Secondly, if the guessed/derived value at any point results in a satisfaction to our condition, i.e. the value leads the function to evaluate below zero, a root finding algorithm would not stop since a root has not yet been found. Instead, a lot of extra time may be spent after this point to find the root, which may actually not exist in some cases.

Since test-data generations are performed at runtime, it is crucial that our model performs the convergence to the needed value as quickly as possible. Consequently, for the purpose of our test-data generation, we define a different technique that takes advantage of existing techniques, yet satisfy our requirement. We refer to this technique as Root Crossing Minimization (RCM), which modifies the Secant and Broyden's techniques. Specifically, since our static analysis do not trace the exact symbolic form of the constraint functions, we are interested in performing all test-data generation numerically, which can be done by applying the Secant/Broyden's methods. RCM however has no particular interest in finding the root. Hence RCM examines the value of the constraint function at each iteration. Should the computation at any point of time determines that the function has a value below the root, then the algorithm terminates since a solution has already been found. Should the algorithm however terminate by actually finding the root where equality is not under consideration, then RCM continues with another phase that initiates an extra vicinity search phase. This vicinity search resorts back to linear programming. Since we are only interested in finding a value that would exactly cross the root, that phase starts by considering the value x_i that led to the root of the function. The algorithm then attempts to evaluate the constraint function with value such as $x_i + \varepsilon$, where ε is a very small value. If this results in finding the root crossing value, then the algorithm terminates successfully. If however the computation results in the constraints function having a larger than zero value, then an attempt is made in the opposite direction; that is with $x_i - \varepsilon$. The process repeats in a similar fashion to the linear test-data generation procedure described in Algorithm 1 earlier in this section. The process terminates either by finding a solution or by exhausting a maximum predefined number of attempts.

It should be noted that there are cases, where no solution can be found even if a root is successfully found. This can happen for instance in cases where coincidentally the function actually touches the root by never crosses it into negative domain. In such cases, there is actually no local solution and hence there is no possibility of finding a usable result by the algorithm. Another similar problem is the oscillations near local minima [124]. In such case, the algorithm may keep in repeating around a local minima without finding a solution even if one exists.

To mitigate such problems, if a solution exists, RCM does not actually terminate once a maximum number of failed attempts is exhausted. Instead, our model treats that as an exhaustion of a maximum number of local attempts. In such cases, the algorithm attempts further exploration of the neighborhood space in an attempt to find a fitter initial candidate. This is done by selecting an initial guess that is marginally different than the initial/prevoiusly used one(s). The actual exhaustion attempts are reached if neighborhood exploration as well as local attempts based on each selected initial candidate from the neighborhood are both exhausted without finding a solution.

Another problem that is being mitigated by RCM, is the division by zero problem. Since our model performs the computation numerically, it first assesses the value of the denominator before computing the approximation of the function derivative. Should the denominator evaluates to zero, the algorithms halts execution, traces the initial guess values, so it can restart the process with different ones. A simplification of our algorithm for test-data generation for non-linear programming is detailed in Algorithm 2.

6.3.3 Execution Management

As discussed in Section 6.3.1, various code instrumentations are applied to the program so that runtime profiling and monitoring can be obtained. Our model accepts both the instrumented program and a suspicious path as input, then attempts to generate data with which the suspicious path is executed. The process of data-generation is however an iterative process, which may require multiple runs of the program before finally a successful generation is achieved. Generally, the generation may generate data that would rather drive the execution towards critical paths. In such case, the execution must be halted immediately, information is collected about the data that causes the failure, and program execution must be re-initiated for further generation attempts. The execution manager of our model is designated to achieve such functionalities. In general, two major components compose the execution manager: monitoring client, and test-data generation server. The decision behind utilizing a client-server architecture for our model is driven by multiple factors. Running both components on the same machine represents a security hazard. In specific, testing a highly/severly vulnerable software may actually compromise the environment where tests are being conducted. Hence, we need a full separation between our environment and the environment where the software is executed. Additionally, since the test-data server is independent of the program being tested, it is possible to utilize it to generate data for more

Algorithm 2 Test-data Generation Algorithm for Non-linear Programming

```
$Input$: instrumented program, suspicious path
$Output$: test-data satisfying the execution of the given suspicious path

while threshold of pre-defined global maximum attempts is not exhausted do
    $i \Leftarrow 0$;
    $x_i \Leftarrow initial\_first\_guess$;
    $c(x_i) = execute(x_i)$;
    if $sucessful\_execution\_of\_suspicious\_path$ then
        $return\ x_i$;
    end if
    // Reaching this point is indicative of execution violating the needed requirements
    $i \Leftarrow i + 1$;
    $x_i \Leftarrow initial\_second\_guess$;
    $c(x_i) = execute(x_i)$;
    if $sucessful\_execution\_of\_suspicious\_path$ then
        $return\ x_i$;
    end if

    // Reaching this point is indicative of failed attempts with the previous two values
    while Threshold of pre-defined local maximum attempts is not exhausted do
        $i \Leftarrow i + 1$;
        // Compute a new value $x_i$
        if $c(x_{i-1}) - c(x_{i-2}) = 0$ then
            Trace history of $x_0$ and $x_1$;
            Halt generation and restart test-data generation process;
        end if
        $x_i \Leftarrow x_{i-1} - \frac{C(x_{i-1})(x_{i-1}-x_{i-2})}{C(x_{i-1})-C(x_{i-2})}$;
        if $sucessful\_execution\_of\_suspicious\_path$ then
            $return\ x_i$;
        end if
        Increment local maximum attempt ctr;
    end while
    // Explore other parts of search space - $\Delta$ is a marginally large offset
    $x_i \Leftarrow x_i + \Delta$;
    Increment global maximum attempt ctr;
end while
```

than one program concurrently.

Execution Management Client

The actual execution of the instrumented program takes place at the execution management client. The client requests test-data generation from the server, then attempts the execution of the suspicious path with such data. If the execution leads to a critical path, the client immediately halts such execution and keeps track of all dynamic relevant information, such as problem nodes and master controlling variables involved in the failure. This information is then communicated back to the server, which uses them to guide the following generations.

Test-data Generation Server

The test-data generation (TDG) server is the entity responsible for generating data. A TCP/IP connection is initially established between the client and the server. Information exchanged between the client and the server drives the generation process. Generally, the following exchanges are performed.

- Suspicious Path Request: As part of the initial static analysis, SVR provides the details of the suspicious path to the TDG server, which stores this information pending on client request to initiate the generation process. Once the client initiates the process, it requests the path details from the server.
- Path Response: The server returns the path information to the client. In particular, the server returns a basic-block representation of the path based on the sequence of function calls made through that path. The representation is similar to the following:

function_name:bb_1, *function_name:*bb_2, ..., *function_name:*bb_n;.

- Data-generation Request: Once the client reaches a statement that requires data generation for a variable, the variable information is provided to the server, which then attempts the generations. Since the program may contain multiple variables with the same name, using block scopes for instance, the information provided by the client must be quite specific and accurate. The client hence provides the full details of the variable including its scope and line number within the program. The format of the request is as follows: *data_request:function_name:*bb_n*:variable_name:line_num*;.

- Generated-data Response: The server attempts to generate data for the requested variable. The data is then sent back to the client to attempt execution with it. The format of the response is as follows: *data_response:function_name:*bb_n*:variable_-name:line_num:value*;.

- Failed-attempts History: The initial attempt of data generation by the server is simply random. Should this attempt fail to provide the proper data, the client detects the execution of a critical path and hence halts the current execution. The client additionally keeps track of various information such as the value of the constraint function at the problem node causing the failure, and the execution history until the failure point. Such information is sent back to the server, which then utilizes them to guide the generation of the following attempts. Such dynamic operation would repeat until either data is generated or a pre-determined threshold of the involved resources is reached. The format of the constraint-function value and the execution history sent by the client are accordingly as follows: *constraint_value:function_name:*bb_n*:value*;

whereas the format of the execution_history is as follows: function_name:bb_1, function_name:bb_2, ..., function_name:bb_n;.

Since the provided failure information from the client is crucial to the server for following generation attempts, the server maintains a set of data structures to record and use such information. Generally, the server maintains the following set of hash tables:

- Branch Direction Table;
- Dependency Table;
- Minimization Table;
- Test-data Generation History Table;

Branch Direction Table

During the static analysis phase, all potential problem nodes (controlling statements) along the path are tracked. A classification is then given to the branches of these nodes, which can be either required or critical. A branch is critical if its execution would permanently drive the program away from the suspicious path. This static analysis information is recorded in the branch direction hash table and is used at dynamic time to monitor the execution and halt the program, should the execution fails to follow the path. The typical contents of the branch direction table is similar to what is shown in Table 6.

Basic Block Number	**Branch Direction**
bb_1:true_branch	required || critical
bb_1:false_branch	required || critical
...	...
bb_n:true_branch	required || critical
bb_n:false_branch	required || critical

Table 6: Branch Direction Table

Dependency Table

During the static analysis phase, all pertinent variables to the suspicious path under test are tracked, which filters out all other variables that have no relation to that path. However, not all of these variables may affect each of the potential problem nodes along the path. Consequently, further static analysis is conducted to determine for each problem node the exact set of variables influencing this node. This information is recorded in the dependency table, and used at dynamic time to determine the exact set of needed generations should the problem node is encountered. Table 7 shows an example of the typical contents of the dependency table.

Basic Block Number	**Input Variable**
bb_1	$x_{11}, x_{12}, \ldots, x_{1m}$
bb_2	$x_{21}, x_{22}, \ldots, x_{2m}$
...	...
bb_n	$x_{n1}, x_{n2}, \ldots, x_{nm}$

Table 7: Dependency Table

Minimization Table

This table keeps track of the values of the constraint functions at each of the problem nodes along the path. The table is maintained at dynamic time. The typical contents of this table is shown in Table 8. Constraint values along the path need to allow for root crossing minimization for successful execution of the path to take place. The recorded constraint value at each of the basic blocks may not necessarily be the appropriate value for the execution to go though the path. An inappropriate value in the table directly indicates that the node is an unsolved problem node, and that this basic block leads to a critical path. These inappropriate values however need to be tracked since the server uses them for further generation attempts.

Basic Block Number	**Constraint Value**
bb_1	val_1
bb_2	val_2
$\ldots$	$\ldots$
bb_n	val_n

Table 8: Minimization Table

Test-data Generation History Table

The server records the values of each attempted generation for each of the master controlling variables along the path. A value from a failed generation, along with the root crossing minimization value of the constraint function at the problem node, are then used by the server to guide the following generation. Should there is a final successful generations for the suspicious path variables, the last generated value for each of them is the final one to be reported. Table 9 shows the typical contents of the test-data generation history Table.

Controlling Variable	**Value**
x_1	$val_{11}, val_{12}, \ldots, val_{1m}$
x_2	$val_{21}, val_{22}, \ldots, val_{2m}$
$\ldots$	$\ldots$
x_n	$val_{n1}, val_{n2}, \ldots, val_{nm}$

Table 9: Test Data Generation History Table

6.4 Hybrid Framework for Automating Security Testing

An overview of our framework and its different components is shown in Figure 31. A description of the security property that need to be tested is injected into the Static Vulnerability Revealer (SVR), along with a representation of the source code of the program to be tested. SVR initiates a set of static analysis as described in Chapter 5, and reports a set of suspicious paths that may actually be vulnerable. Further static analysis is then conducted to achieve various functionalities. Dynamic analysis phase is then initiated to generate concrete test-data to prove the existence of the violation. A description of the different components of our framework is given below. Various transformations of the code take place first. Instead of working directly on the source code, an object-oriented representation of the GIMPLE representation is constructed and manipulated to provide data-flow and control-flow information of the suspicious path. Code instrumentations are also conducted on the GIMPLE representation of the program to allow execution monitoring of the program. The instrumented program along with the flow information of the suspicious path are then passed to the execution manager, which performs the data generation.

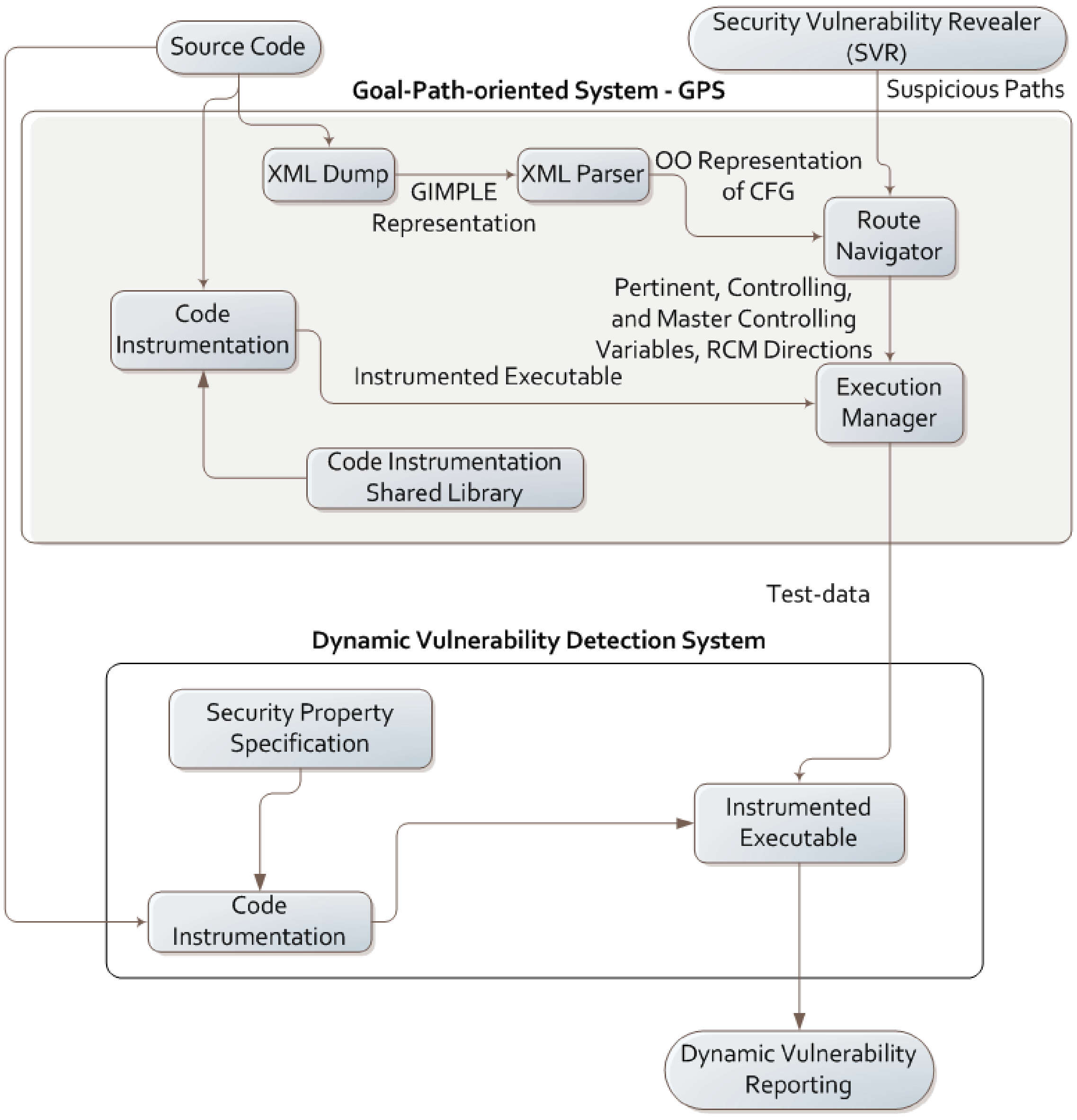

Source Code
Security Vulnerability Revealer (SVR)
Goal-Path-oriented System - GPS
Suspicious Paths
XML Dump
GIMPLE Representation
XML Parser
OO Representation of CFG
Route Navigator
Code Instrumentation
Pertinent, Controlling, and Master Controlling Variables, RCM Directions
Instrumented Executable
Execution Manager
Code Instrumentation Shared Library
Test-data
Dynamic Vulnerability Detection System
Security Property Specification
Instrumented Executable
Code Instrumentation
Dynamic Vulnerability Reporting

Figure 31: Overall Design of Hybrid Security Testing System

- XML-Dump: This component accepts the source code of the program, in specific the GIMPLE representation of the code, and produces an XML representation of the code. Such a representation facilitates the manipulation of the source code.

- XML-Parser: This components accepts the XML representation of the program and composes an equivalent object-oriented representations of it.

- Static Analyzer: This components accepts both a suspicious path from SVR, as well as the GIMPLE object-oriented representation of the program. Further static analysis is conducted by this component where information about control-flow of the suspicious path is tracked.

- Code Instrumenter: This component performs the needed instrumentations for the dynamic data generation phase. In general, this component performs instrumentation for input abstraction, history tracking and code monitoring, as described in Section 6.3.1.

- Code Instrumentation Shared Library: The code instrumenter injects monitoring code into the program. Some of these injections are simply method calls that will be executed if the call is made. This component contains the implementation of these monitoring APIs.

- Execution Manager: An instrumented version of the program is finally provided to this execution manager component, which actually controls and performs the data generation. The test-data generation client and server are implemented as parts of this component.

6.5 Conclusion

We have presented in this chapter a hybrid approach to test-data generation, which takes advantage of the powerful capabilities of both static and dynamic analyses. This novel approach is a Goal-Path-oriented, which initially starts by the security analysts precisely specifying the security properties that they wish to test the software against. The approach conducts static analysis to determine the set of all suspicious paths that the software may have in violation to the given property. The approach then conducts dynamic test-data generation, and monitoring process to produce concrete data with which the vulnerability is proven, hence eliminating false positive reports. Prior to conducting the dynamic analysis, various further static analysis processing are still needed and must take place first, including code instrumentation, and control-flow analysis.

We have presented our approach to test-data generation, which considered important issues such as linear and non-linear programming, root finding, function minimization and the problem of constraint solving. In this context, we have distinguished between solving linear and non-linear functions and presented our root crossing minimization approach for non-linear data generation. Since the process of generating data is conducted at dynamic time, where time aspect is crucial, we have has investigated issues such as immediate halting of failed executions and exploration of search space in an attempt to find different candidates if the initially guessed ones do not lead to a solution after a bounded number of attempts.

We have designed and implemented a framework in support to our approach. The framework provides capabilities for the automation of security testing. That includes automated detection of potential vulnerabilities that may violate security properties, as well as an automated test-data generation to prove the real existence of such vulnerabilities. We have detailed the different components of the framework to illustrate what type of analysis is being used at each component as well as how these components work together to automate security testing. To analyze our approaches, and show that they are indeed better than other approaches, we have conducted various experiments on our system. The details of such experiments and their results are detailed in Chapter 7.

Chapter 7

Integrated System and Experiments

7.1 Introduction

This chapter presents our integrated system for security testing. In that context, we introduce the structure, interfaces and functionalities of our system and its various components. We also provide the details of the experiments conducted over the system to validate our approaches. Prior to providing the details of our full integrated system, we first highlight the design and implementations of some of the major components composing our framework. Section 7.2 provides the design details of our Team Edit Automata model. Section 7.3 provides the details of our code instrumentation in support to team edit automata. Section 7.4 provides a brief view of our APIs for program monitoring, which are needed for dynamic test-data generation. Section 7.5 provides the details of our integrated system and its user interfaces. Section 7.6 provides the details of the experiments conducted on our system and the results of such experiments. Finally, in Section 7.7, we provide a summary and conclusion of the subjects covered in this Chapter.

7.2 Team Edit Automata - Design Overview

As discussed in Chapter 3, a Team Edit Automaton (TEA) is composed of a set of interacting Component Edit Automata. We use C++ for the implementation of TEA. In general, four major C++ classes compose our design of TEA and its components, namely `ComponentAutomata`, `TeamAutomata`, `Event` and `SuggestionSolver`. The following subsections, 7.2.1 and 7.2.2, provide further information on the implementation of `ComponentAutomata` and `TeamAutomata` respectively.

7.2.1 Component Edit Automata - Implementation

Since each component represents a specific security property based on the security analyst testing goals, the `ComponentAutomata` class is provided as an abstract class and is to be inherited by a concrete class defining the exact property. To facilitate the security analyst task, we additionally provide a graphical interface, with which the analyst can rather draw the automata instead of writing its object-oriented code, as explained in Section 7.5.

Each component automaton belongs to a team automata. The association between these automata is made through a private member of the team, namely `_team`. To provide full flexibility, we allow a component to switch between different teams. This is made through the utilization of the `UnregisterToTeam()` and `RegisterToTeam()` interfaces, which allow the component to unregister itself from its current team and registers with another one respectively.

In our implementation, we allow minimal coupling between the component automata itself and the state transition functionality. The idea behind such design decision is to enable any implementation of a finite state machine to be incorporated into our hierarchy structure. A component automaton reacts to events through the use of the `QueryEvent()` and `ExecuteEvent()` interfaces. The `QueryEvent()` interface results in the automaton responding to the event by merely producing a suggestion of an execution based on this event. The execution of the actual state transition is then made through calling the `ExecuteEvent()` interface.

7.2.2 Team Edit Automata - Implementation

A Team Edit Automaton manages multiple Component Edit Automata. The `RegisterComponent()` and `UnregisterComponent()` interfaces are used to allow such association.

The team also manages all events in the system. All unprocessed events are inserted into an event queue (`_events`). Events are generally classified as external or internal events. Internal events are those events sent between the different components, and hence not visible to the outside of the system. These events are enqueued through the utilization of the `AddInternalEvent()` interface. The team additionally provides an interface, namely `Query()`, to the monitored program. In particular, the instrumented monitoring routines periodically call this interface to collect team-wise suggestions. Upon the reception of this call, a request is sent to all the component automata to collect their suggestions. The decision of how these suggestions are evaluated for a final action to be taken is left to the `SuggestionSolver` component. Once a final team-wise suggestion is determined, the suggestion is transmitted as an output action to all involved components, which result in the appropriate state transitions of these automata. Figure 32 shows the flow of operations initiated by the reception of the event.

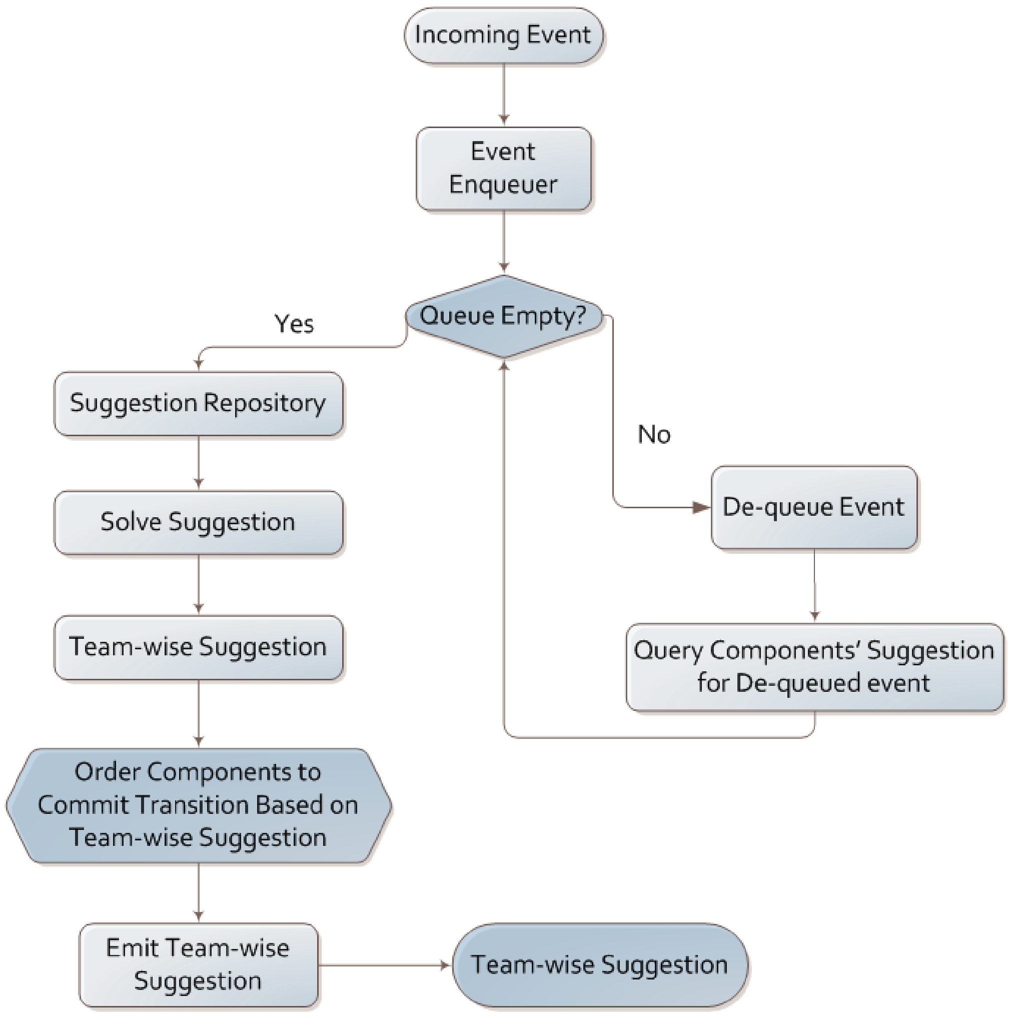

Incoming Event
Event Enqueuer
Queue Empty?
Yes
No
Suggestion Repository
Solve Suggestion
Team-wise Suggestion
Order Components to Commit Transition Based on Team-wise Suggestion
Emit Team-wise Suggestion
Team-wise Suggestion
De-queue Event
Query Components' Suggestion for De-queued event

Figure 32: Flow of Operations for an Incoming Event

7.3 Team Edit Automata - Code Instrumentation

In Chapter 4, we have discussed code instrumentation and its major techniques. We have then introduced our novel AOP-compiler-assisted code instrumentation technique, which combines both powerful capabilities of AOP pointcut model and the compiler-assisted approach. Major factors influence our decision to utilize the compiler-assisted approach. Program structures are naturally known to compilers, hence compilers know the lexical structure and semantics of the code being compiled. Additionally, they are capable of building a more structured representation of the code (i.e. Abstract Syntax Tree (AST)) and Control Flow Graphs (CFG). Consequently, instrumentation can be performed very precisely at any program point.

Another advantage is related to execution time. Higher execution performance can be achieved as a result of utilizing compilers. This issue is crucial to dynamic analysis, where multiple executions of the program may take place before vulnerabilities can be detected. In addition, since compilers perform program optimization, they are capable of optimizing the instrumented code as well.

Our decision to use the compiler-assisted approach, had to be followed by another decision on which compiler to use. The answer to the latter question was GCC. Our decision was driven by many factors. Beside the fact that GCC is well-developed and has been extensively used and tested in multiple platforms, the compiler supports compilation of various programming languages such as C, Objective C, C++, Java, Fortran, and more. Moreover, GCC uses a universal intermediate language, called GIMPLE, to represent programs written in different languages, including C, C++, Objective-C, Fortran, Java, Ada, and Go [64]. Consequently, by instrumenting at the GIMPLE level, support for various language can be achieved.

However, since GCC does not provide the needed functionality to support our security instrumentation, an extension to the compiler is needed. In fact, multiple extensions are needed to support different programming languages. Considering the security issues of the C languagee, we choose to implement an extension to the GCC compiler for C. The details of GCC internals as well as our GCC extension to support code instrumentation have been provided in Chapter 4.

7.4 Test-Data Generation - APIs for Dynamic Monitoring

For the purpose of test-data generation, further code instrumentation is required for the purpose of monitoring the generation process as well as the dynamic program execution. The instrumented code is injected at sensitive program points to allow automation of the generation process, collection of information upon failure to generate the needed data, as well as to monitor the execution itself. We implemented the instrumented routines as calls to APIs, which are injected into the proper program points for data generation. The following briefly describes the provided APIs for test-data generation monitoring purposes:

- `_initialize()`: Performs various initializations needed for test-data monitoring process, such as allocating needed memory and creating log files.
- `_finalize()`: Releases all allocated resources for the monitoring process.
- `_do_execution_monitoring(int bb_index, const char* funcName)`: Upon data generation, program execution is attempted with the intent to execute the vulnerable path. This API call would halt program execution should the execution is driven towards a critical path.

- `_calculate_constraint(int lhs, int rhs, int oper, int bb_-index, const char* funcName)`: A call to this API is needed so that the constraint value is calculated prior to the execution of the conditional statement.

- `_abstract_input(const char* format, ... )`: This API is needed for automation purposes. The API would abstract program actions, such as user input, by replacing it with function calls. These calls would then perform the needed operations automatically without user interaction.

- `_send_msg(char* msg)`: Enables the client of the test-data generator to send messages to the server;

- `_receive_msg(char* msg)`: Enables the client to receive messages from the server.

7.5 Integrated System Overview

This section provides an overall view of our intergraded system. In that context, we present the system's interface, and describe its functionalities and capabilities.

7.5.1 System Interface

The Graphical User Interface (GUI) of the system provides multiple tabs, which are controlled by the analysts to utilize the desired functionality. Figure 33 provides a general overview of the system when the *Project Overview* tab is selected.

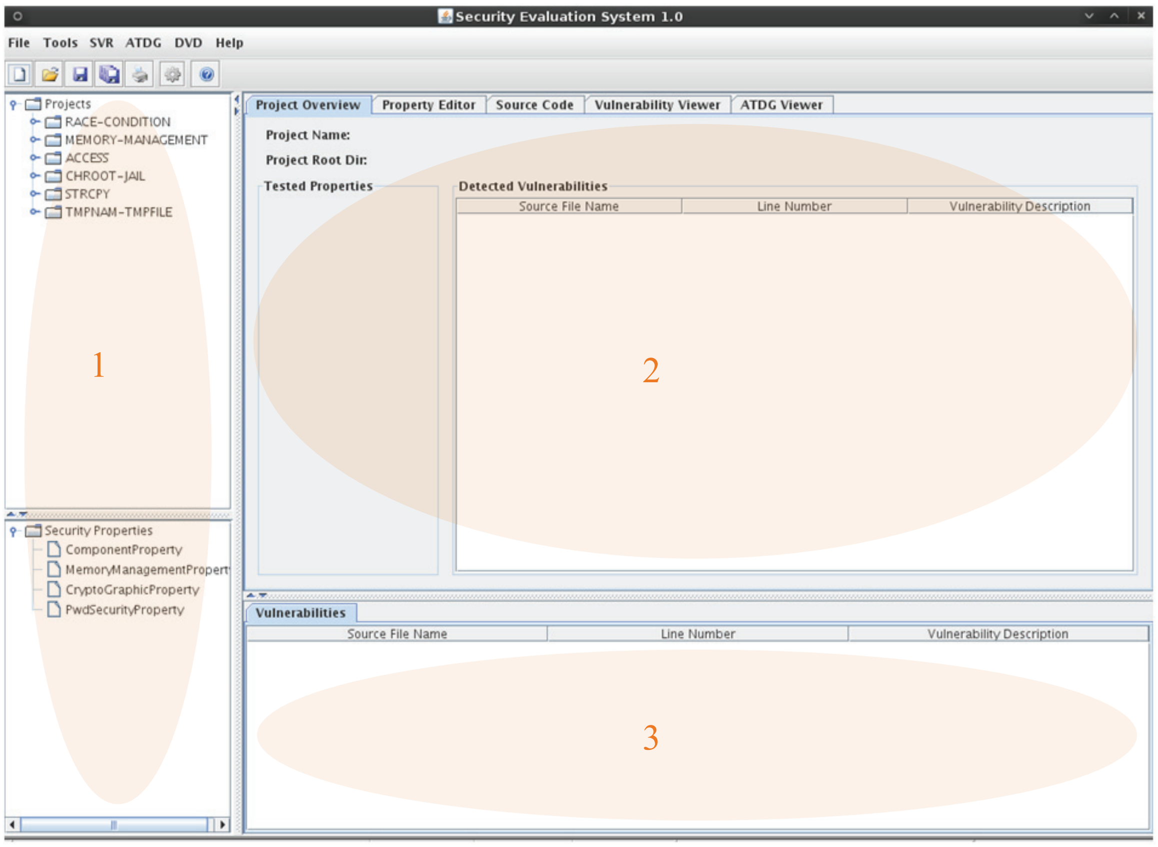


Figure 33: Integrated System - Overview

As seen in Figure 33, three main components compose the interface. Area number 1 allows the analyst to select one, or multiple, projects for security testing. Area number 2 represents the main window of the interface. Area number 3 is designated for the reporting of detected vulnerabilities. We will shortly describe the utilization and functionality of each of the tabs, but prior to that we first explain how the system is to be configured for use.

7.5.2 System Configuration

For the security analyst to use the system for the first time, there are few simple configurations that must be performed first. This configuration must be done by the analyst since they are system-dependent, and so automation is not advisable. The desired configurations are set through the *File –> Preferences* menu. Figure 34 and Figure 35 illustrate this operation.

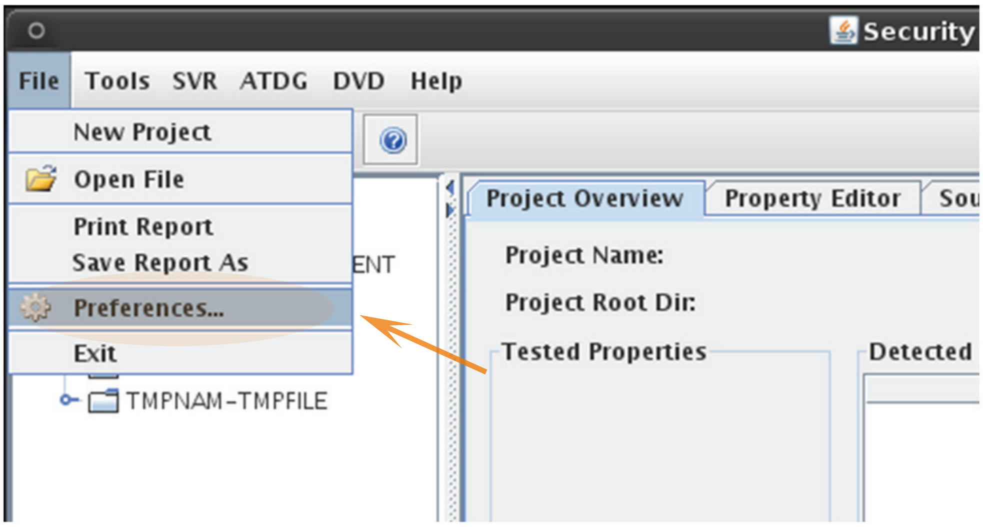


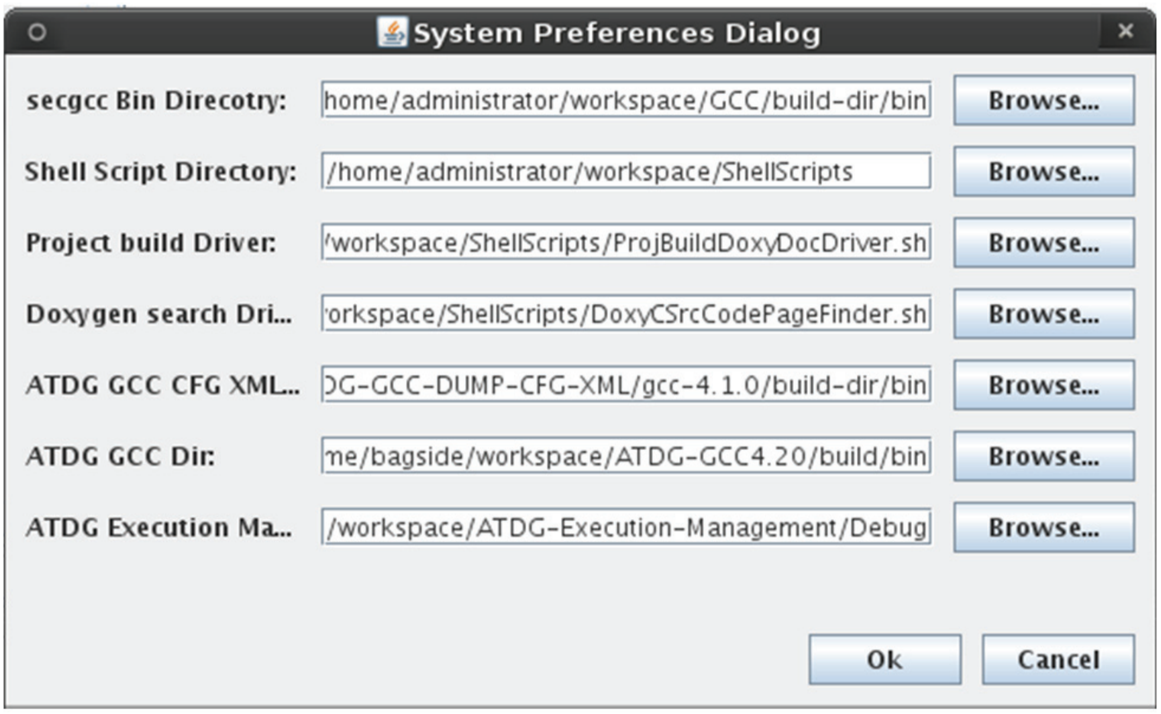
Figure 34: System Settings through *File –> Preferences* Menu

7.5.3 Project Management

To facilitate the use of our tool, we design it as an Integrated Development Environments (IDE), similar to other common software applications. The analyst needs to create a new project then provides the root directory where the source code to be tested is located. The creation of the project results in our system navigating the root directory and displaying all files in this directory in the Project component of our interface. Figure 36 and Figure 37 show how these operations are performed.

7.5.4 Security Property Specification

Once the project is opened, security analysts are capable of specifying the security property that they wish to test the software against. The property is specified as an automaton as described in Chapter 3. The analyst is capable of specifying the property by writing


System Preferences Dialog

secgcc Bin Direcotry:	home/administrator/workspace/GCC/build-dir/bin	Browse...
Shell Script Directory:	/home/administrator/workspace/ShellScripts	Browse...
Project build Driver:	'workspace/ShellScripts/ProjBuildDoxyDocDriver.sh	Browse...
Doxygen search Dri...	'orkspace/ShellScripts/DoxyCSrcCodePageFinder.sh	Browse...
ATDG GCC CFG XML...	DG-GCC-DUMP-CFG-XML/gcc-4.1.0/build-dir/bin	Browse...
ATDG GCC Dir:	me/bagside/workspace/ATDG-GCC4.20/build/bin	Browse...
ATDG Execution Ma...	/workspace/ATDG-Execution-Management/Debug	Browse...

Figure 35: Preference Dialog

its object-oriented code. While this task may not be that difficult for experienced programmers, it certainly represents some sort of overhead. To avoid such overhead, our tool allows the analyst to rather specify the automata by graphically drawing it. Our tool then compiles this graphical representations and automatically produces the object-oriented representation of it. Figure 38 shows how the automaton can graphically be stated by drawing it in the Property Editor window of the interface, which is displayed when the Property Editor tab is clicked.

The analyst is simply capable of defining new states by selecting the *edit* mode ( ) then drawing the state. The attributes of this new state, such as its name, can then be modified if needed by using the *Attribute Viewer* component of the interface, as shown in Figure 39.

The transitions are added by simple mouse movement and clicking, similar to other

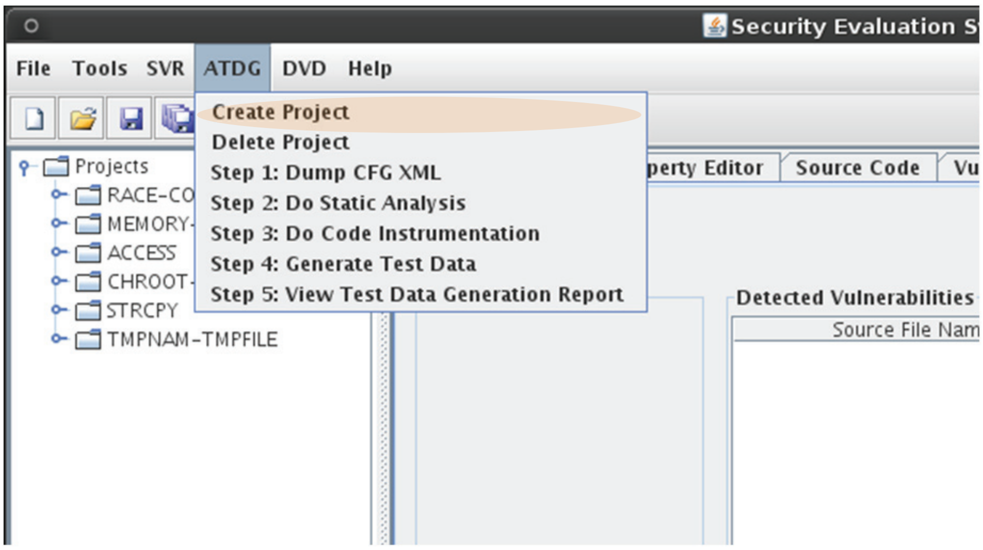


Figure 36: Project Management Menu

traditional graphical software. For instance, the analyst can add a transition between two states by mouse clicking the originating state, moving the mouse to the destination state then releasing it. Our system allows multiple transitions to be drawn between two states, or a transition to take place from one same state to itself, which reflects the reality of state machines.

Following the creation of a transition, it may be needed to define events or guards for this transition. Our system allows that by utilizing the *Event* and *Guard* tabs of the *Attribute Viewer*. Figure 40 and Figure 41 show how these operations are performed.

Transition Actions As described in Chapter 3, a component edit automaton must emit an action; such as suppress, halt, etc. The analyst must hence set this action. This is done through the utilization of the *Action* tab of the *Attribute Viewer* component. Figure 42 shows this operation.

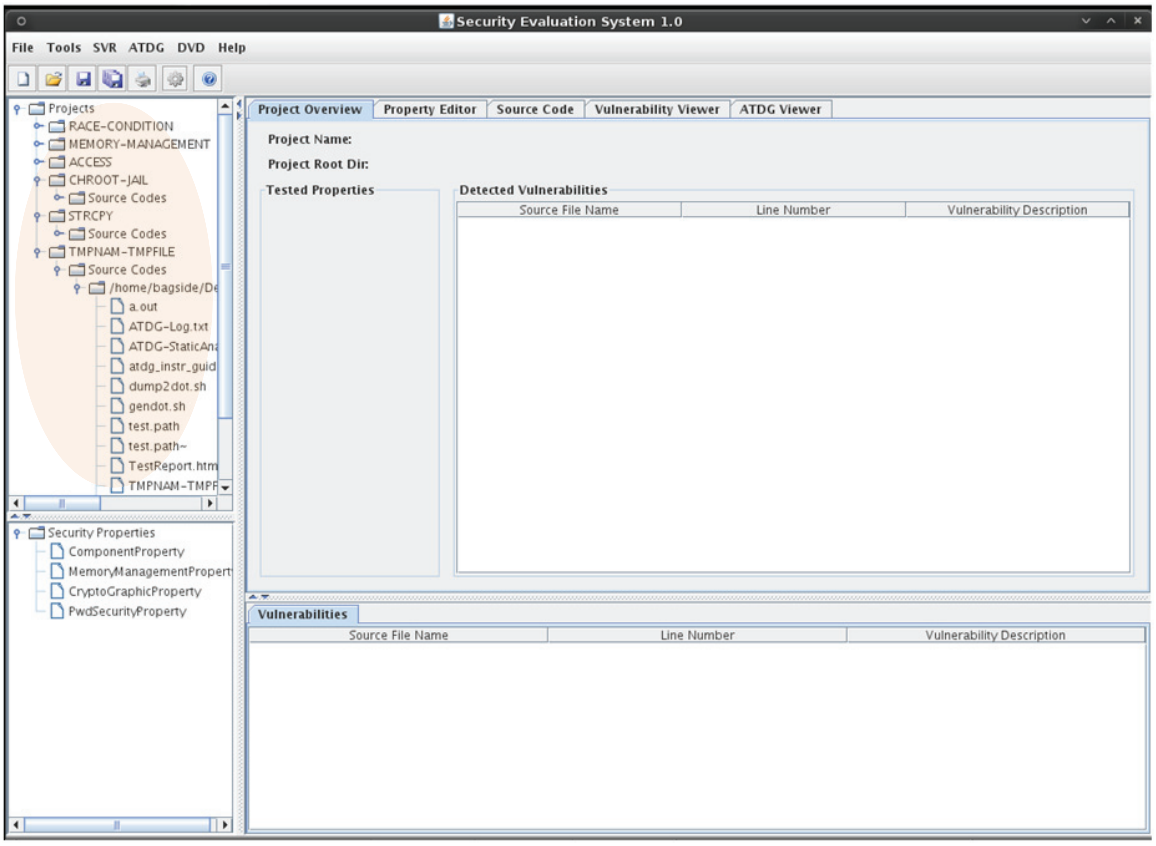


Figure 37: Opening Project for Security Testing

Saving Security Properties Once the analyst finalizes the security property and saves it, using the button, a series of operations are initiated by the system as follows:

1. All graphical components of the interface are stored, so they can be restored when the the property is reloaded later on. To ensure precision of such operation, we actually serialize all the components to XML representations.

2. A .sm (state machine) file is generated. This file reflects the various transitions of the automaton.

3. The graphically-drawn automaton is compiled into an object-oriented code representation of the property.

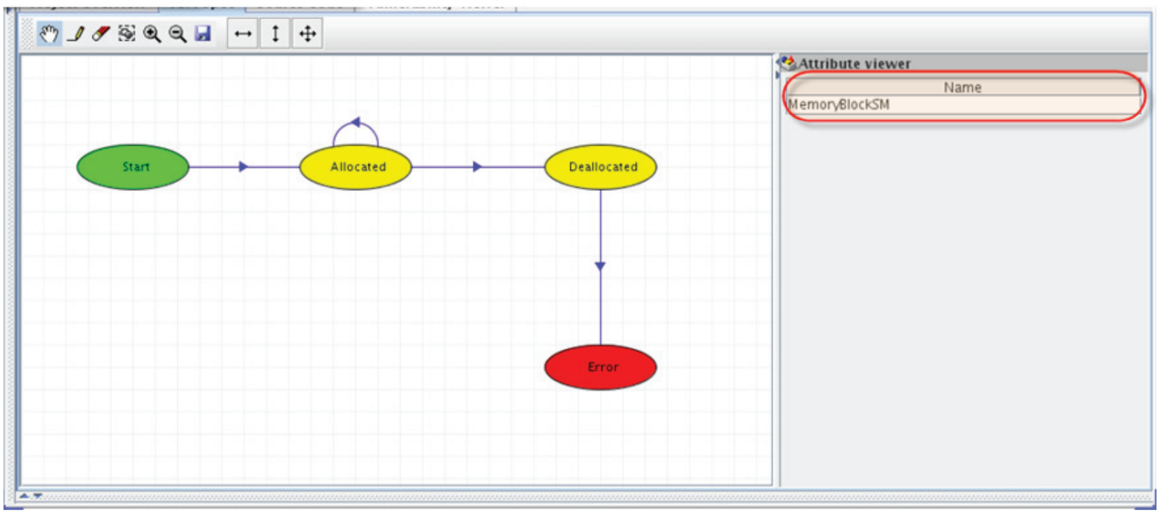


Figure 38: Specifying a Security Property Graphically

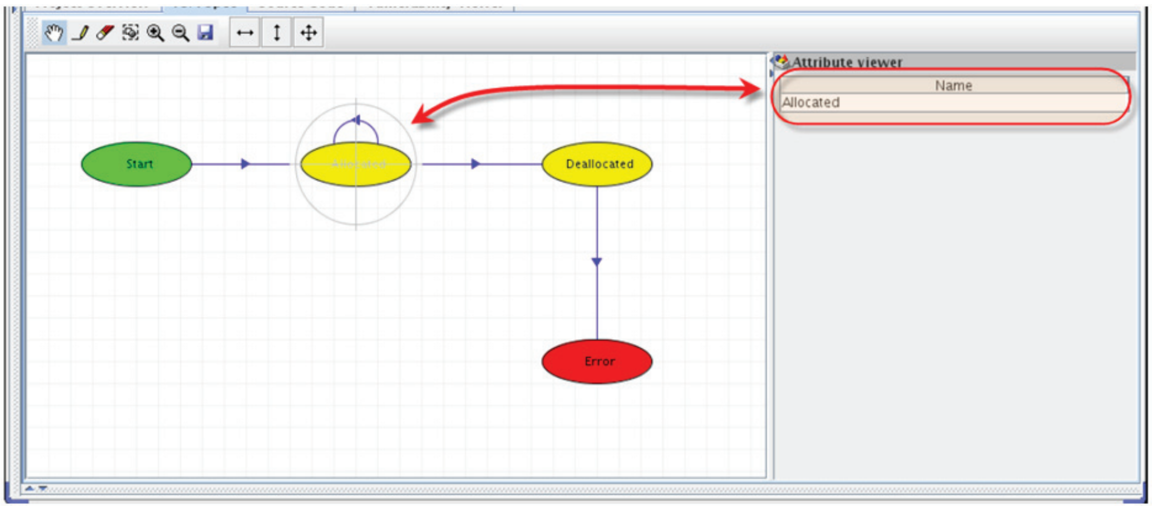
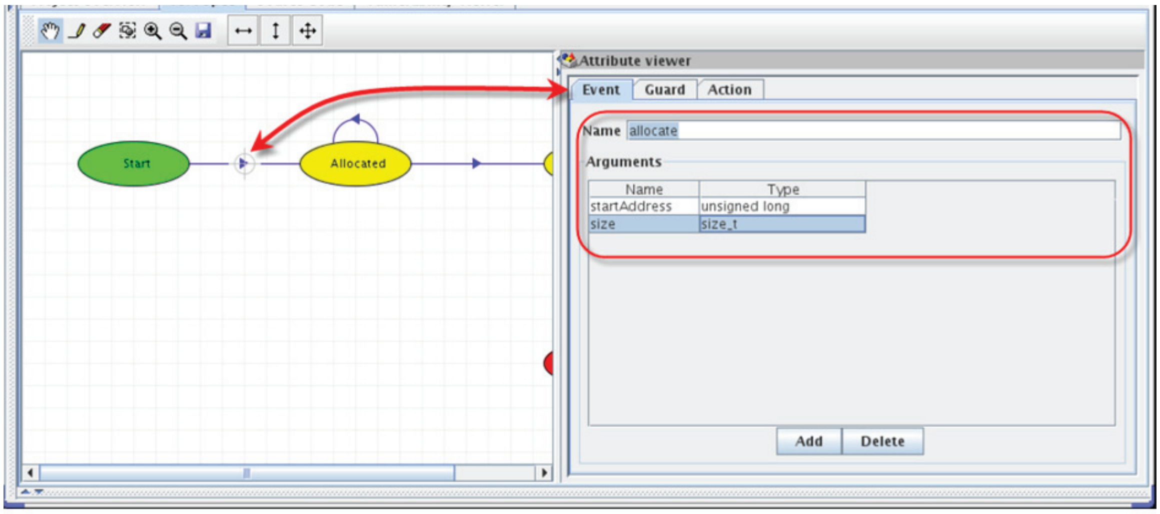


Figure 39: Modifying State Attributes

4. An automatic generation of a concrete class representing the automaton is performed. The class is created as an extended/child class of `ComponentAutomata`.

7.5.5 Building Projects

Once the security property is specified, the analyst can build the project then executes it through the utilization of the appropriate provided menu items to build and execute the

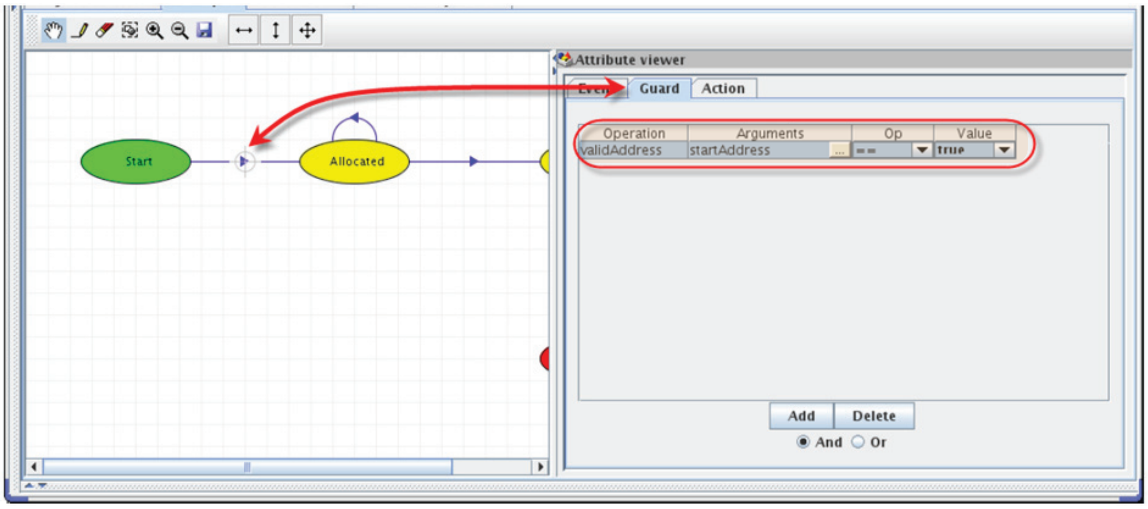


Name: allocate

Arguments

Name	Type
startAddress	unsigned long
size	size_t

Figure 40: Setting Transition Events



Operation	Arguments	Op	Value
validAddress	startAddress	==	true

Figure 41: Setting Transition Guards

project. The analyst is capable however of building the project without specifying any properties. In such case, a basic built of the project is conducted without any code instrumentation taking place.

Our system automates project building through the utilization of a shell script that calls a `Makefile`. The `Makefile` must hence be located. By default, our system assumes the existence of that file in the root directory of the project. If this is not the case, then the

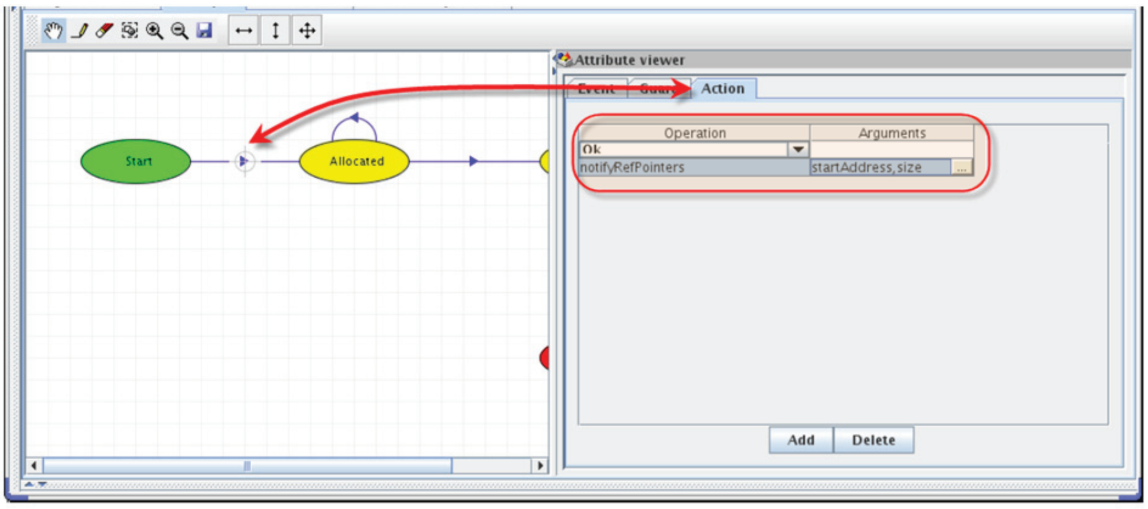


Figure 42: Integrated System - Setting Transition Actions

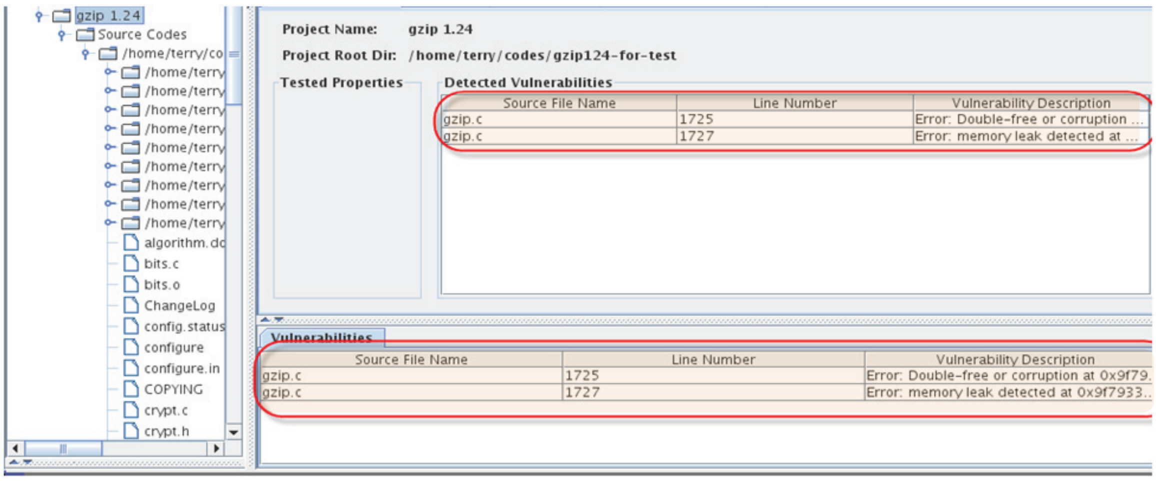
analyst needs to specify the location of that file through the re-configuration of the system.

7.5.6 Project Execution and Vulnerability Reporting

Upon a successful built of the project, the analyst is capable to execute the project to obtain the vulnerability reports, if any. The execution of the project is initiated through the utilization of the proper menu for such action. Once the execution is conducted, error reports and dynamically generated test-data are reported for the analyst to view. Figure 43, Figure 44, and Figure 45 show examples of such reports.

7.6 Experiments

We conduct various experiments over our system. These experiments are carefully designed to analyze and test a broad range of concerns. Beside testing the basic functionalities of the system, we particularly need to analyze the detection power of our system, as well



Project Name: gzip 1.24

Project Root Dir: /home/terry/codes/gzip124-for-test

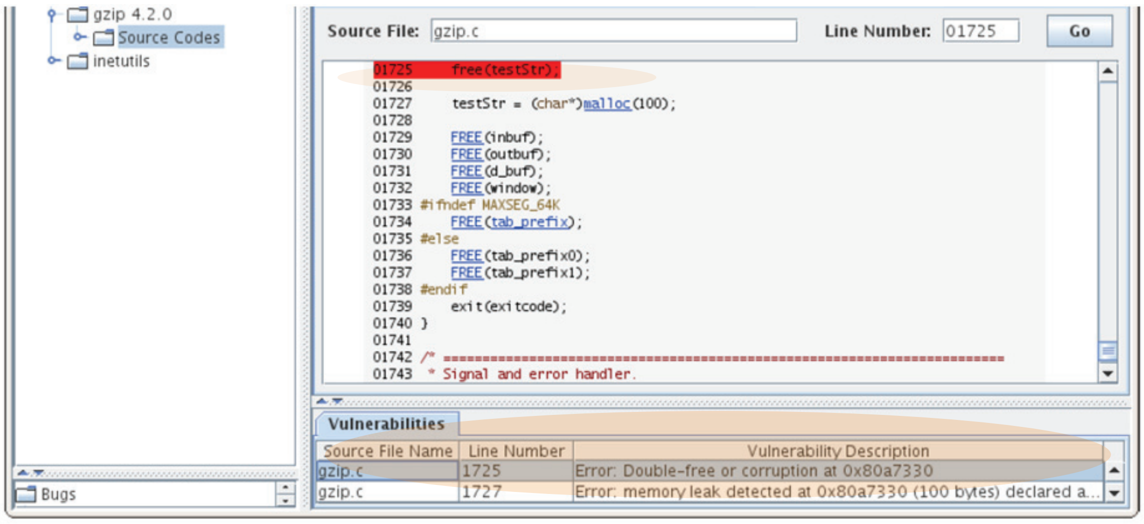
Detected Vulnerabilities

Source File Name	Line Number	Vulnerability Description
gzip.c	1725	Error: Double-free or corruption ...
gzip.c	1727	Error: memory leak detected at ...

Vulnerabilities

Source File Name	Line Number	Vulnerability Description
gzip.c	1725	Error: Double-free or corruption at 0x9f79.
gzip.c	1727	Error: memory leak detected at 0x9f7933..

Figure 43: Reports of Detected Vulnerabilities



Source File: gzip.c **Line Number:** 01725

```
01725     free(testStr);
01726
01727     testStr = (char*)malloc(100);
01728
01729     FREE(inbuf);
01730     FREE(outbuf);
01731     FREE(d_buf);
01732     FREE(window);
01733 #ifndef MAXSEG_64K
01734     FREE(tab_prefix);
01735 #else
01736     FREE(tab_prefix0);
01737     FREE(tab_prefix1);
01738 #endif
01739     exit(exitcode);
01740 }
01741
01742 /* ===========================================================================
01743  * Signal and error handler.
```

Vulnerabilities

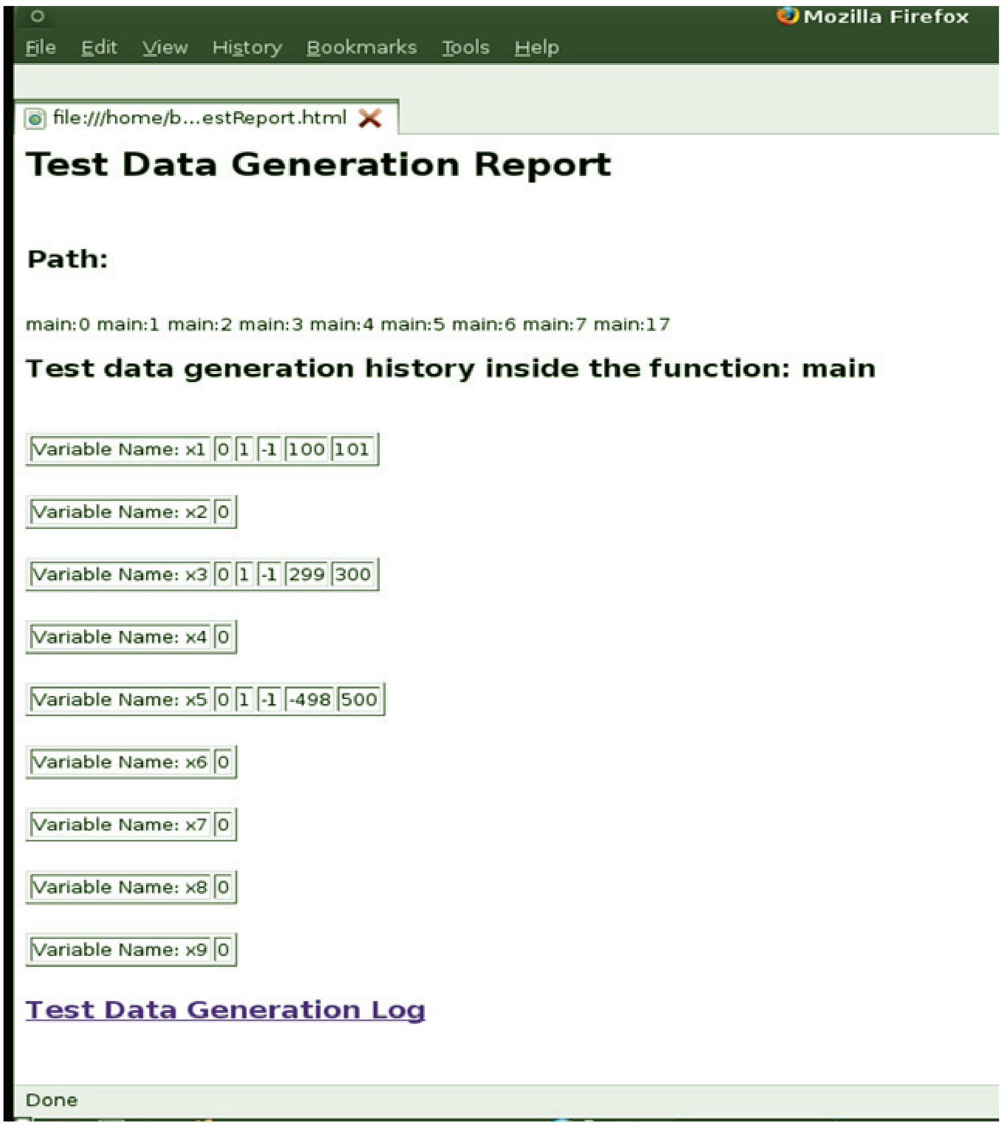
Source File Name	Line Number	Vulnerability Description
gzip.c	1725	Error: Double-free or corruption at 0x80a7330
gzip.c	1727	Error: memory leak detected at 0x80a7330 (100 bytes) declared a...

Figure 44: Code View of Vulnerabilities Reported in Figure 43

as the capability to detect both low-level vulnerabilities and high-level vulnerabilities. Additionally, we need to analyze other important aspects such as performance and scalability. The following subsections provide the details and the results of our experiments.

7.6.1 Experiments Environments

The system experiments are conducted under the following environments:


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///home/b...estReport.html

Test Data Generation Report

Path:

main:0 main:1 main:2 main:3 main:4 main:5 main:6 main:7 main:17

Test data generation history inside the function: main

Variable Name: x1	0	1	-1	100	101
Variable Name: x2	0				
Variable Name: x3	0	1	-1	299	300
Variable Name: x4	0				
Variable Name: x5	0	1	-1	-498	500
Variable Name: x6	0				
Variable Name: x7	0				
Variable Name: x8	0				
Variable Name: x9	0				

Test Data Generation Log

Done

Figure 45: Test-Data Generation Reporting

- **Operating System:** Ubuntu Linux release 6.10;
- **Linux Kernel:** Linux kernel version 2.6.17-11-generic;
- C/C++: GCC 4.2;
- **Java Runtime Environment:** Java(TM) SE Runtime Environment (build 1.6.0_02-b05);

- **Shell:** GNU bash version 3.1.17(1)-release;
- **autoconf:** GNU Autoconf version 2.60;
- **make:** GNU Make 3.81;
- **Awk:** mawk version 1.3.3;
- **Doxygen:** doxygen version 1.4.7;
- **Moped:** Moped version 2.

7.6.2 Testing Against Low-Level Security Vulnerabilities

The main objective of these experiments is to test the detection power of our system against low-level security properties. In that context, the experiments target the following low-level vulnerabilities:

- Pointer de-referencing - use of wild and null pointers
- Double-freeing of pointers
- Freeing wild and null pointers
- Memory leak
- Reading uninitialized/unallocated memory
- Memory/Buffer overflow
- Out-of-boundary array indexing

The reason behind our choice to target these specific low-level vulnerabilities is due to the fact that they are quite common in C/C++ programs. Additionally, these types of vulnerabilities have always been the cause of major attacks. Another reason is that these types of vulnerabilities have been addressed by other security detection tools, so we are able to provide detection-power and performance comparisons with these tools as well.

Tested Programs

To perform the desired testing, we implemented a set of C programs, with known low-level memory management vulnerabilities. In particular:

- Program 1: This program include pointer double-freeing vulnerability. The program particularly includes two circular referencing pointers, on which one of them has been freed.
- Program 2: This program includes buffer-overflow vulnerabilities. The program allows for a buffer size to dynamically be set through user input. Further manipulation of the buffer is then performed without proper verification of its size.
- Program 3: This program includes out-of-boundary array indexing vulnerability. The program allows for the array manipulation through an index variable. The value of that variable is improperly modified before accessing the array, hence resulting in the vulnerability.
- Program 4: This program includes buffer-overflow vulnerability. The vulnerability results from an improper use of the buffer, which results in manipulating it beyond its storing capacity.

- Program 5: This program targets false-positive reporting. The program included a pointer double-freeing vulnerability that is impossible to execute. Consequently, the program has no exploitable vulnerabilities.

- Program 6: This program includes buffer-overflow vulnerability. The main cause of the vulnerability in this program is the use of an unsafe method provided by the standard C library, resulting in buffer-overflow of a static-size buffer.

Experiments Results

The programs are tested using our tool as well as other two well-known security testing tools, Klocwork K7.5 and Insure++ 5.1, to provide us with a benchmark for compression. Our choice of these two particular tools for comparison is intentional. As part of this research, we have evaluated various similar tools and based on this evaluation we consider Klocwork and Insure++ to have powerful capabilities compared to other static-analysis and dynamic analysis tools accordingly. The results of the tests are detailed below and summarized in Table 10.

- Program 1: Our system successfully detects the double-freeing vulnerability in this program. Additionally, following the first freeing operation of the pointer, the system suppresses the second freeing action, avoiding the program from crashing. Insure++ is also able to detect the vulnerability, however the program execution ended in crashing after reporting an assertion failure problem. Klocwork on the other hand fails to detect the vulnerability.

- Program 2: Our system successfully detects the vulnerability reporting an illegal memory access attempt, once the buffer is overflown. Insure++ also detects the vulnerability but fails to identify its exact cause. Klocwork fails to detect the vulnerability and reports no errors.

- Program 3: Both our system and Insure++ successfully detects the exact vulnerability of that program. On the other hand, Klocwork cannot detect this vulnerability. In that particular case, it is very possible that the lack of detection by Klocwork is due to its static analysis nature where the data-flow affecting the index variable may have not been considered.

- Program 4: Our system successfully detects the vulnerability reporting an illegal memory access attempt, once the buffer is overflown. Due to version incompatibility of the compiler, we are unable to instrument or compile the program with Insure++, hence we cannot obtain any results about Insure++ behavior. As for Klocwork, it fails to report any vulnerabilities.

- Program 5: We execute this program multiple times using different values that controls its flow. Our system does not report the vulnerability, since it is actually not exploitable. Again, we are unable to compile this program with Insure++, and hence cannot obtain any concrete results of its behavior. On the other hand, Klocwork detects and reports the vulnerability in the code, which is clearly a false-positive reporting.

- Program 6: Our system successfully detects the vulnerability once the C method is called, hence detecting the exact location and the cause of the vulnerability. Insure++

reports an unknown error, while Klocwork reports no errors at all.

In conclusion, these experiment results show that our system, through its underlying static/dynamic techniques, is effectively capable of detecting low-level security vulnerabilities. As the results also indicate, some of these low-level security vulnerabilities are difficult to detect through mere static analysis approaches.

7.6.3 Testing Against High-Level Security Vulnerabilities

This group of experiments is designed to validate our system functionality and detection capability against high-level security vulnerabilities. To be more precise, we target the detection of a mixture of both low-level and high-level vulnerabilities listed by the United States Department of Homeland Security [45]. In that context, the experiments targeted the following high-level vulnerabilities:

1. Temporary File Utilization and Access Privileges:

   This type of vulnerabilities, known as TEMPNAM-TMPFILE [45], is often caused by improper use of intermediate and temporary files needed at program execution. Software programmers usually use this type of programming approaches, where they create intermediate files to temporarily store information and allow different program entities to communicate at execution time. While the approach itself might be considered as reasonable at a first glance, it is vulnerable. The calls to create a temporary file actually result in the creation of a file name, instead of a file. An attacker may take advantage of this by creating the file itself after the call is made but before the actual file is created. Since the contents of the file are totally under the attacker's

Vulnerable Programs		Experiment Results	
Program #	Vulnerability Details	Used Tool	Result Summary
Program 1	Double-freeing	Our System	Error: Double-free or corruption at $0x80a68b0$ at Experiment1.c::17
		Klocwork K7.5	Passed: 0 Errors, 0 Warnings, 0 Filtered
		Insure++ 5.1	Freeing dangling pointer, 1 unique occurrence at an unknown location
Program 2	Buffer-overflow - dynamic setting	Our System	Error: Illegal memory access at $0x80a7990$ at Experiment2.c::20
		Klocwork K7.5	Passed: 0 Errors, 0 Warnings, 0 Filtered
		Insure++ 5.1	INSURE_ERROR: Internal error, 1 unique occurrence
Program 3	Out-of-boundary array indexing	Our System	Error: Illegal array indexing with index value -2 at Experiment3.c::15
		Klocwork K7.5	Passed: 0 Errors, 0 Warnings, 0 Filtered
		Insure++ 5.1	Writing array out of range: staticBuf[i] Index Used: 2. Valid range: 0 thru 9 (inclusive)
Program 4	Buffer-overflow - improper buffer use	Our System	Error: Illegal memory access at $0x80a68fb$ at Experiment4.c::20
		Klocwork K7.5	Passed: 0 Errors, 0 Warnings, 0 Filtered
		Insure++ 5.1	Could not compile with Microsoft Visual C++ & Insure++ 5.1
Program 5	No Exploitable Vulnerabilities	Our System	(No errors were reported)
		Klocwork K7.5	Experiment5.c(25: Severe: Double freeing of freed memory pointed by 'buf2'
		Insure++ 5.1	Could not compile with Microsoft Visual C++ & Insure++ 5.1
Program 6	Buffer-overflow	Our System	Error: Illegal memory access at $0x80a908d$ at Experiment6.c::19
		Klocwork K7.5	Passed: 0 Errors, 0 Warnings, 0 Filtered
		Insure++ 5.1	INSURE_ERROR: Internal error, 1 unique occurrence

Table 10: Experiment Results of Low-Level Security Vulnerabilities Testing

control, it is possible to attack the system, for instance by destroying/modifying the application's data. Worst, the standard C library provides a family of calls, such as `tmpnam()`, `tempnam()` and `tmpfile()` that facilitate the creation of temporary files. However these calls are considered to be unsafe. The names of the created files using these methods can easily be guessed by attackers, hence facilitating such attacks even further.

Consequently, a property against this type of violation would target multiple high-level concerns: the utilization of unsafe calls to create the file, the use of temporary files, as well as access privileges to these file. Our automaton description of this security property is shown in Figure 46.

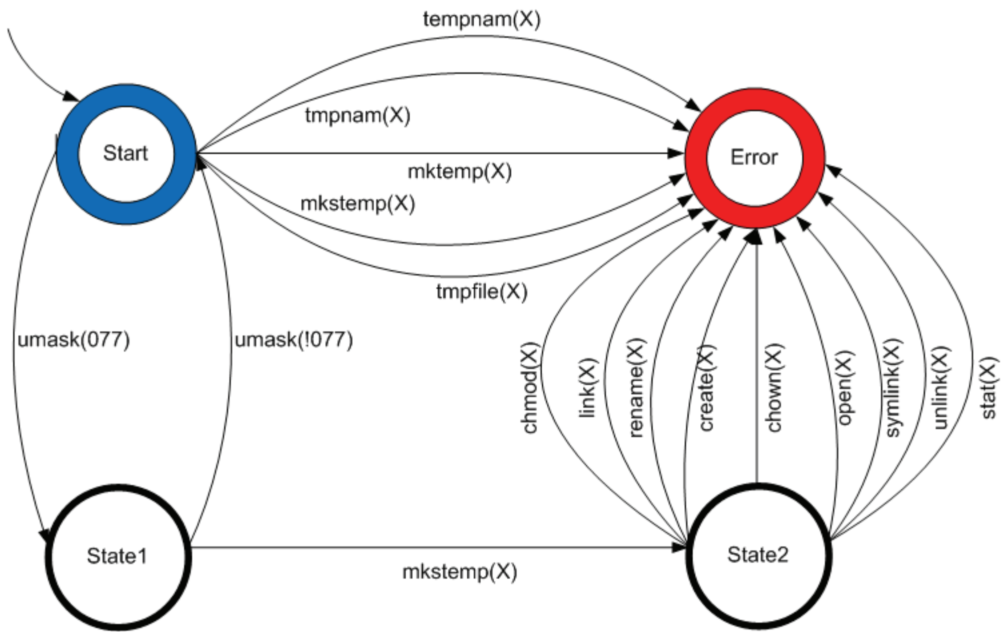


Figure 46: Temporary File Utilization and Access Privileges Automaton

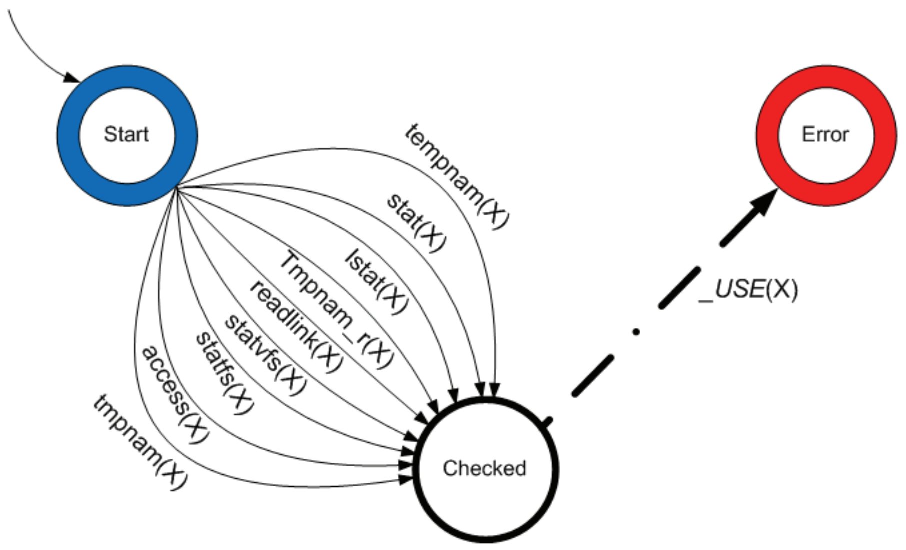
The property specifies the following: The use of the unsafe calls from the standard C library would directly transit the automaton to the *Error* state. Prior to calling an appropriate method to create the file, such as `mkstemp()`, a call to `umask(077)` should be made, restricting access permissions over these files only to the owner. A change of that mask after being set would simply transit the automaton back to its *Start* state. Assuming that a sequence of proper calls is made, a transit is made to a state where the creation of temporary files is allowed. However, executing any call from this latter state that would change access privileges, such as `chown`, `chmod`, etc., would result in transiting the automaton to the *Error* state.

2. Synchronization and Race-Conditions:

   This type of vulnerabilities is known as Time-Of-Check-to-Time-Of-Use (TOCTOU), which is caused by synchronization problems, where a process checks for a particular condition before performing an action. For instance, CPU interrupts between verification and use may allow multiple processes to concurrently proceed to critical sections of the code. Attackers can use these potential timing gaps between verification and use/proceeding further, to cause very severe attacks, such as modifying application's data or producing incorrect execution behavior. Figure 47 shows our automaton representation of this security property.

3. Unauthorized Access to System Resources and Privilege Escalation:

   This is rather a common type of security vulnerabilities, where an attacker gains unauthorized access to some system resources, then uses this access to harm the system. To mitigate some of these problems, operating systems may provide some methods to properly restrain access to system resources. The `chroot()` system



_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, scandir, symlink, system, t_open, truncate, umount, unlink, utime, utimes, utmpname

Figure 47: Race-Condition Automaton

call is one of such methods provided by the UNIX operating system. The main purpose of `chroot()` is to restrict the access of a user process to a particular portion of the file system, hence preventing access to unauthorized resources. The method performs this task by establishing a virtual root directory for the calling process and restrains access of the process and its children to that directory. Consequently, the directly is commonly known as "chroot jail". However, `chroot()` suffers various serious problems. The method cannot defend against intentional improper manipulation by privileged *root* users. On most systems, `chroot()` contexts do not stack properly. For instance, a jailed program with sufficient privileges may perform a second `chroot()` to break out [145]. The call hence must be followed by a call to `chdir(/)` to prevent the breakage-out-of-jail problem. Another problem with `chroot()` is that it requires a *root* privileges to be called. A program that fails to reset the privilege back after the call is made, could open a great opportunity for attackers to take advantage of this improper privilege escalation.

Consequently, the method must be used with care; otherwise its utilization may actually compromise systems security due to illegal access and improper privilege escalation. Figure 48 shows our automaton representation of `chroot()` utilization property.

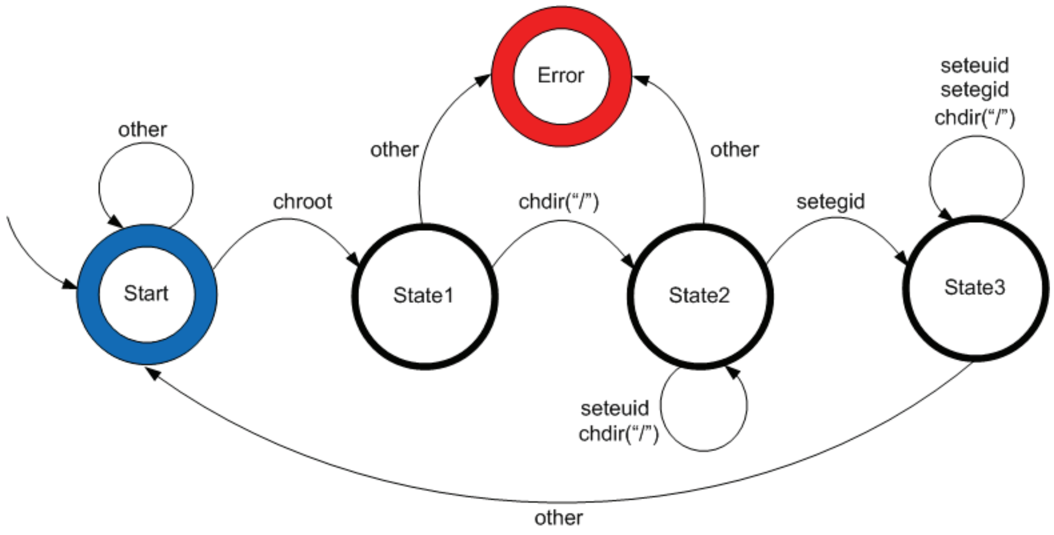


Figure 48: CHROOT-JAIL Automaton

4. Utilization of Vulnerable Methods from Standard Libraries:

   Standard libraries provided with programming languages offer a large set of useful methods that facilitate programmers' tasks. For instance, the standard C library provides various methods that enable programmers to easily work with strings and buffers. However, some of these methods have been found to be quite vulnerable and the source of very severe attacks [143]. An example of such methods is `str-cpy()`, which allows a source buffer to be copied into a destination buffer. Due to potential difference in size between the source and the destination, the method may result in buffer-overflow, leading to various vulnerabilities including denial-of-service attacks. In general, buffer-overflow is one of the most common causes of security attacks. Instances of such include AT&T Win Virtual Network Computing (WinVNC) [1] vulnerabilities, where an attacker generates a custom-crafted HTTP request and sends it to the WinVNC server. The server will overflow and overwrite variables on the stack, including the return address. The Cisco 7*xx* Router Password Buffer Overflow Attack [33] is another example, where an attacker only needs to have a remote prompt to enter a user-name and a password to login into the router. A long string for the password is then entered, which causes a buffer-overflow and results in crashing the router. The router crash consequently breaks down the network. A small script is then written by the attackers, which waits for the router to restart, then repeats the attack, hence resulting in a denial-of-service attack on the network.

To avoid such attacks, the destination buffer must be guaranteed to be of a larger size than the source, or at least of an equal size. Additionally, the terminating character, (`'\0'`), must be present in the copied string. Figure 49 shows our automaton representation of buffer-overflow using `strcpy()` method from the standard C library, where *n* is the size of the destination buffer.

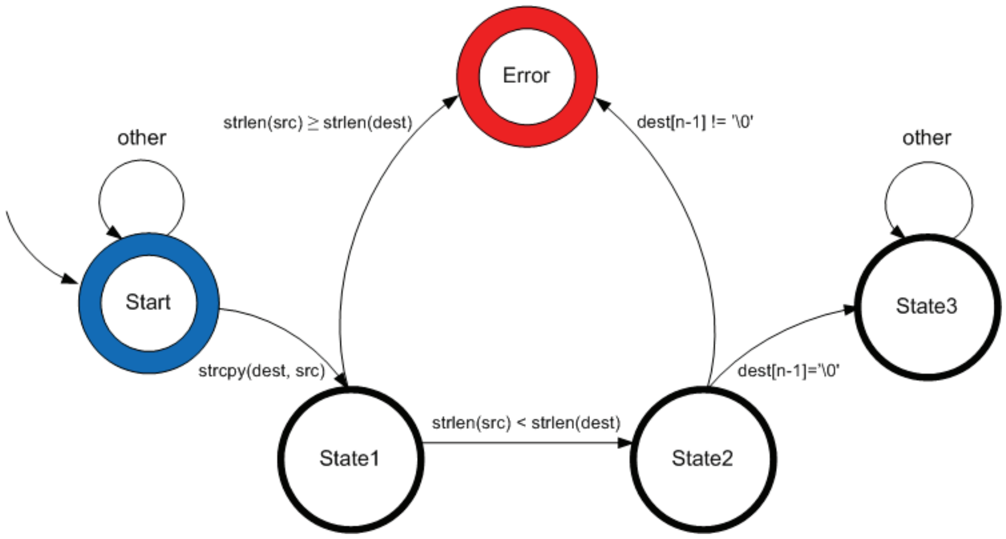


Figure 49: STRCPY Automaton

5. Memory Management:

   Improper pointer manipulation and memory allocations are among the very common sources of vulnerability attacks, especially in languages such as C/C++. That is due to the powerful capabilities given by such languages to programmers to create and manipulate pointers easily through the "$\star$" operator. We focus in this experiment on testing the major causes of memory attacks, which are as follows:

   - Use of unallocated pointers: Most programming languages separate the issue of creating pointers from allocating space to these pointers. In other words, it is syntactically legal to create a pointer without initializing it. Since pointers are naturally created to facilitate movement around memory areas, an uninitialized pointer may actually include the address of a previously allocated memory location. This consequently comes with a hefty price in terms of security since there is no guarantee that a pointer will indeed be pointing to an allocated memory prior to its use.

- Memory leak: This is rather a serious problem that may result in consuming a large amount of the executable memory of the system, leading the system to a standstill. Languages such as C and C++, do not maintain automatic garbage collectors to claim previously allocated memory that is no longer in use. Instead, these languages require the program to delete any allocated memory that is no longer needed. Programmers may very well overlook such needed operations, resulting in memory leak.
- Double-free: This is rather an extreme opposite operation to the memory leak problem. With such operation, programmers delete allocated memory as needed, but later follow with another deletion to the same memory location. The double deletion may not necessary be made through the same pointer. The second deletion may actually be made through a different pointer, that is pointing to the same location.

Figure 50 shows our automaton representation of memory management, which targets all the aforementioned memory management vulnerabilities.

Tested Programs

The main goal of this part of the experiment is to analyze the detection capabilities of our system against high-level security vulnerabilities, as well as the effectiveness and the performance of our test-data generation techniques. A test suit is designed with five programs carrying various vulnerabilities, which we refer to as Program 1 to Program 5. Below is a

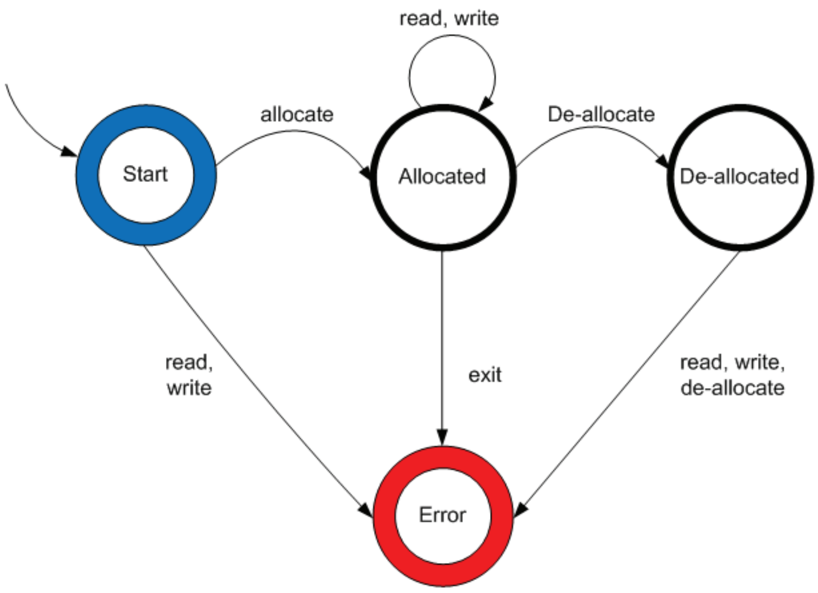


Figure 50: MEMORY-MANAGEMENT Automaton

description of the exact violations of these programs.

- Program 1: It contains violations of synchronization and race-condition security properties.
- Program 2: It contains violations of privilege escalation and unauthorized access. In particular, the program utilizes UNIX `chroot()` system call improperly.
- Program 3: It targets violations in relation to memory management. In specific, the program improperly utilizes C pointers and memory allocation/deletion.
- Program 4: It contains buffer-overflow vulnerabilities. The program improperly utilizes the `strcpy()` method.
- Program 5: It contains violations related to the utilization of temporary files.

Experiment Results

In relation to the detection capabilities, our system is successfully able to detect the vulnerabilities on each of the five programs. The vulnerabilities are reported as a set of suspicious paths, on which we then perform test-data generations. The generation of test-data eliminates the reporting of our system to any false-positives.

Since our experiments additionally target both the effectiveness and the performance of our test-data generation techniques, we need to obtain similar results using other test-data generation techniques for comparison reasons. Hence, four different results are obtained. The first set of results is obtained using random-testing since it generally provides the benchmark for other comparisons. The second set is obtained using our Moped reachability checker, while switching off our slicing mechanism for the data generation process. The third set is obtained using Moped reachability checker with our slicing mechanism being enabled. The final set is obtained using our hybrid static-dynamic analysis approach. Table 11 shows the results obtained using Random testing. Table 12 shows the results obtained using Moped reachability checker with and without slicing being enabled. Table 13 shows the results using our hybrid approach for test-data generation.

Program name	**Security property**	**Time (milliseconds)**	**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 11: Experimental Results using Random Testing

As shown in Table 11, the random approach is able to generate the needed test-data for

Program name	Security property	Time without slicing (milliseconds)	Time with slicing (milliseconds)	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 12: Experimental Results using Moped Reachability Checker

all five programs. The amount of time and the number of iterations needed to generate the data vary greatly from one program to another, which is expected due to the nature of this approach. Nonetheless, we are able to obtain our comparison benchmark. We particularly notice that the needed time to randomly generate the data is greatly affected by the existence of an equality comparison at the problem nodes of the suspicious paths.

Program name	Security property	Time (milliseconds)	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 13: Experimental Results using the Hybrid Approach

Table 12 provides the results of our test-data generations using Moped with and without slicing. The results also show that a great enhancement is obtained using this approach in comparison to random testing. While the results are promising and show that this approach provides tremendous advantage over random testing, they have to be analyzed with a great care; otherwise they can be quite misleading. Due to the static nature of this approach, it has many serious drawbacks. The approach is not suitable for use with large and complex software. While performance seems impressive for small-size software with limited state space, it dramatically and quickly degrades once the state space increases. Additionally, the complexity of the program, such as the number of variables involved and the high coupling between different components, directly, and greatly, influence the performance of the approach, even if the state space is limited. The coupling between components also affects slicing quite negatively that the improvement using slicing becomes negligible. This approach is hence very appropriate for use with relatively smaller-size software such as embedded software.

Figure 13 shows the results obtained using the hybrid approach. For practical reasons and due to the vast difference in capabilities and applicability between this approach and the Moped reachability approach, these results can meaningfully only be compared to random generation. The obtained results conclude that the hybrid approach not only allows for a better generation performance but also a consistent one. The needed time to generate the data is dependent on various factors, such as the number of problem nodes along the suspicious path, the number of executions of the program until data is generated, the number of linear and non-linear constraint functions along the path, etc. In spite of these factors, the results show that the generation time is still reasonable, and that an increase of the state space would increase such generation time in a linear fashion.

In conclusion, we consider the hybrid approach as the most promising approach for test-data generation. That is due to its generation performance, its ability to work on a large spectrum of software including those complex ones with large state space, and its mitigation to false-positive reporting.

Project Name	Num of Files	Num of Lines	Project URL
Amaya v9.55	2281	907947	`http://www.w3.org/Amaya/`
gdLibrary v2.0.35	214	107889	`http://www.libgd.org/`
gzip v1.24	96	24247	`http://www.gzip.org/`
inetutils v1.5	498	206574	`http://ftp.gnu.org/gnu/inetutils/`

Table 14: FOSS Projects used in Scalability and Usability Experiment

7.6.4 Scalability, Usability and Automation

This group of experiments targets the validation of other crucial aspects. In that context, we need to analyze the capability of the system for testing large-size software. Additionally, we need to analyze its level of automation; in other words, how much involvement is actually required by the analysts to use our system.

To achieve the needed goals, we conduct the experiment over four open-source software projects of various sizes; gzip version 1.24, Amaya version 9.55, inetutils version 1.5, and gdLibrary version 2.0.35. The details of these projects are shown in Table 14. We apply code instrumentation over these projects for detecting memory management vulnerabilities.

Experiment Results

- gzip version 1.24:

  The initial project creation and configuration are performed as expected following the usual procedure described in Sections 7.5.2 and 7.5.3. The code instrumentation is preformed successfully without any difficulties. The manual intervention needed is minimal, where the creation of a makefile for building the project is needed, which is achieved through the execution of *Autoconf* [2], an automatic configuration utility.

- Amaya version 9.55:

The initial project creation and configuration are performed as expected following the usual procedure described in Section 7.5.2 and Section 7.5.3. The code instrumentation and building of the project however required further user intervention. Our system is designed to validate software where source code is available, such as Free and Open-Source Software. Should the software requires the utilization of additional libraries, these libraries must be available. Amaya represents an example of a FOSS project that uses a particular library that is unavailable in the operating system. The library needs to be installed, which is achieved by manual intervention.

In general, testing this project requires three operations to be performed manually. The required library has to be installed. A distribution-specific directory has to also be created. Finally, a command-line configuration option has to be stated for proper execution of the makefile for building the project. Nonetheless, regardless of this manual intervention, we are able to successfully conduct the needed experiment over the software.

- `gdLibrary` version 2.0.35:

  Our selection to test this particular project, which is a library, is aforethought, since we need to validate the automation level of our system for such type of software. As anticipated, due to the particular nature of this software, more effort is initially needed to create the project. A makefile for building and instrumenting the project is needed, which is simply achieved through the execution of autoconf. However, the execution of this project requires additional user intervention effort due to the fact that it is only a library. Nonetheless, we are able to successfully conduct the needed experiment on this software.

- `inetutils` version 1.5:

  The project creation and configuration are performed as expected following the usual procedure described in Section 7.5.2 and Section 7.5.3. The code instrumentation is preformed successfully without any difficulties. The manual intervention needed is minimal, where a makefile needs to be created, which was again achieved through the execution of autoconf.

In summary, the experiments conclude the following about our system capabilities concerning scalability, usability and automation level:

- Project creation and configuration are quite straightforward to conduct. They are performed in a standard way similar to utilizing other common software. Specifying the security properties, and the final reporting by the system are quite easy as well, from user-utilization perspective.

- The automation level of the system is reasonably high when the validation process is concerned with open-source executable projects. As detailed above, the system mainly requires minimal user intervention in three of the four experiments to create a makefile through an automatic configuration utility. No modification to the makefiles are needed in all three cases. For the fourth experiment, Amaya, further manual intervention is needed in order to build the project due to the fact that the software required a library that was unavailable in the system. So far, our system is unable to provide further automation for such cases since library dependency is project-specific.

- Our experiment over gdLibrary is indicative of the difficultly achieving full automation. In such particular cases, as an instance, where the validation is not performed over an executable project, rather over a library, further manual effort is required. In such cases, dynamic analysis of the library behavior may be needed before finally be able to analyze it. This consequently requires manual intervention by the analyst. While this intervention may not be that complex, it is definitely indicative of the difficultly and limitations achieving full automation.

7.7 Conclusion

In this chapter, we have presented our integrated system, which combines the various components and approaches presented in this thesis for software security testing and vulnerability detection. In that context, we have first detailed the design and implementation of some of the major components composing the system. An introduction to the system and its user interface have been provided. Finally, we have detailed the various conducted experiments over the system for the purpose of analyzing it and comparing it with different approaches.

A set of experiments has been conducted over the system to provide a real-life analysis of the system and its different approaches. Three sets of experiments have been carefully conducted. The experiments have targeted various aspects including the detection against low-level and high-level security properties as well as the effectiveness of test-data generation. The experiments have additionally targeted other crucial aspects such as scalability, usability and system automation. The results have concluded that both our Moped reachability approach and the hybrid approach are far more effective than random generation.

Careful analysis of the results has concluded that the Moped reachability checker approach is most suitable for testing relatively small-size applications, such as embedded software. The results have showed that the hybrid approach is capable of effectively providing consistent test-data generation.

In terms of automation, the experiments have concluded that our system is reasonably capable of providing a quite high-level of automation in the more-general case when the software being tested is executable. The experiments have also concluded that in other particular cases, such as testing a library, a more manual effort and a user involvement by the analyst may be required.

Chapter 8

Conclusion

Software is taking an increasingly predominant rule in today's world. Further, the utilization of open source software in particular is exponentially on the rise. However, a great deal of software has been designed, implemented and deployed without having security in mind. Such software are currently in usage within environments that are security-critical including networking/worldwide web. Software attacks still make first-page headline news. Various vulnerabilities are only discovered after they have been exploited and in many cases already resulted in sever damages. Software testing is a primary approach for alleviating many of these security problems. Additionally, if the source code is accessible, such as in open source software, then effective testing can be conducted. However, testing is generally a very comprehensive process that can be quite time and cost consuming. In this context, the automation of software security testing of open source software, which is addressed thoroughly in this thesis, is becoming a very challenging and interesting problem.

In the sequel, we summarize briefly the main thesis contributions:

- Security specification language, with which security analysts are effectively capable of expressing the security properties they wish to test the software against.
- An Aspect-oriented and compiler-assisted approach for code instrumentation.
- New Aspect-Oriented pointcut extension for AspectC++ to support security testing.
- Extension to the GNU compiler collection (GCC) to enable the desired security code instrumentation.
- Reachability analysis approach for vulnerability detection.
- Test-data generation approach through static and reachability analyses.
- Hybrid approach for vulnerability detection and test-data generation.

Although the current approaches for software security testing may be effective in revealing some security vulnerabilities, they suffer shortcomings caused by their underlying analysis. For instance, static-based approaches suffer the major drawback of false positives. Dynamic-based approaches suffer significant drawbacks such as the potential of unrealistically excessive number of executions before detection may be achieved. Another critical concern is the matter of automation. Manual intervention may have devastating consequences in terms time, cost and reliability, especially when testing a large-scale software. Consequently, the amount of manual involvement and the dependence on a particular analysis to achieve what is needed are very important. These issues have been addressed in the proposed vulnerability detection approaches for security testing.

The realization of the proposed approach has been achieved by elaborating various interrelated components. A security specification language called Team Edit Automata

(TEA) has been proposed to enable security analysts to precisely specify the needed security property that need to verify the software against. The language allows the specification of individual properties as well as relating these properties into one team so that a deterministic behavior is achieved if more than one property is concurrently violated. The language also allows the analysts to alter program execution if desired through the insertion of additional actions or the suppression of others. All detected flaws are finally reported by the team.

The proposed approach to property specification requires the instrumentation of monitoring code into the original code to be tested. Consequently, an approach to code instrumentation is needed. We have investigated the applicability and usability of Aspect-Oriented Programming (AOP) for such purpose. Our research concluded that the current implementation of AOP is incapable of allowing it to be used for such desired purposes. Consequently, we propose the needed extensions to the language, particularly to AspectC++, with which the language is capable of achieving what is desired for code instrumentation. Nonetheless, understanding the benefits of the pointcut model of AOP, we propose a novel approach to code instantiation, which is based on the compiler-assisted instrumentation and the AOP pointcut model. For the compiler, we choose the GNU Compiler Collection (GCC). The proposed approach requires extending GCC first in order to support the needed security instrumentation. Considering that C is among the most vulnerable languages, in terms of security, we implement an extension to the GCC compiler for C.

In order to achieve the desired security testing results, other major issues have to be considered. These are the matter of test-data generation and the dependence on the underlying analysis. These issues have been addressed by elaborating two different approaches for security testing. The realization of the first approach has led to the proposal of reachability analysis for detecting reachability security properties, where the execution of some, possibly orderly, events is sufficient to violate a security property. The approach is additionally heavily based on sole static analysis. The realization of the proposed approach has been achieved by elaborating a static analyzer, referred to as static vulnerability revealer (SVR), and a test-data generator based on program slicing and the Moped checker. The static analyzer is automatically identifies vulnerable points in a control-flow graph. Given a source code of a program, 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 illustrate the feasibility and capability of such approaches by implementing an entire framework based on such approaches. Our results show that approaches that are heavily based on static analysis can be very useful and more suitable for testing relatively small software, such as embedded systems. For testing large systems, we need to have other approaches and tools that combine the powerful capabilities of both static analysis and dynamic analysis. This has led to the proposal of a more generic approach that is based on a synergy between both static analysis and dynamic analysis. We refer to such approaches as hybrid static-dynamic analysis approaches.

The realization of these propositions has been achieved by implementing a general framework for automating security testing, including security vulnerability detection and test-data generation. We propose a novel approach to test-data generation, namely the Goal-Path-oriented System (GPS). Both data-flow and control-flow analyses are initially performed by our approach. Following this static-analysis phase, a dynamic analysis phase is performed for the purpose of generating concrete test-data. The execution of the program with such data would result in the execution of the vulnerability, hence eliminating false positive reopenings. To validate our proposed approaches, we implement a framework, which provides capabilities towards the automation of security testing. A set of experiments has been conducted over the system to provide a real-life validations of the system and its different approaches. The obtained results conclude that our approaches are effective in detecting both low-level and high-level security vulnerabilities. In terms of automation, the experiments have concluded that our system is reasonably capable of providing a high-level of automation in the more-general case when the software being tested is executable. In other cases, some level of analyst intervention seems to be unavoidable, which raises the question of whether full automation of security testing, where no manual involvement whatsoever is required, is practically achievable. This very well represents one of the major issues that could be considered for future work.

Limitations and Future Work

The work presented in this thesis can be extended in different directions. Currently, our framework and approaches are limited to non-GUI software. Additionally, we have focused on C language in particular due to its high susceptibility to security vulnerabilities. The future extension plan of our work will target the systematic integration of security testing approaches at the early life cycles of software development. This would effectively disallow/reduce the deployment of many vulnerabilities. Additional work is also needed to increase the expressiveness of team edit automata by addressing issues such as timeliness. We also need to address further limitations of the AOP technologies for security testing code instrumentation through elaborating new pointcut and primitive constructs. In particular, propose needed extensions to other AOP languages, such as AspectJ. We also need to formalize the proposed instrumentation behaviors as primitives in the pointcut/advice model of AOP. An additional effort is also required to revisit the current prototype design and implementation to deal with the issue of automation as well as security testing of GUI-based software, software libraries, etc.

List of Publications

The following is the list of publications derived from the thesis work:

Journal Papers

1. R. Charpentier, M. Debbabi, A. Hanna and TFOSS Research Team. Security Evaluation and Hardening of Free and Open Source Software (FOSS). In the Electronic Communications of the EASST, Volume 33, 2010.

Conference Papers

1. A. Hanna, H. Z. Ling, X. Yang, and M. Debbabi. A Synergy Between Static and Dynamic Analysis for the Detection of Software Security Vulnerabilities. In the Proceedings of the Interational Symposium on Information Security (IS'2009), Vilamoura, Algarve-PortugalLNCS, Volume 5871, Pages 815-832, 2009, Springer Verlag.

2. A. Hanna, H. Ling, J. Furlong, and M. Debbabi. Towards Automation of Testing

High- Level Security Properties. In the Proceedings of the DBSEC-2008 Conference, July 13-16, 2008, London, England, Lecture Notes in Computer Science, Pages 268-282, Springer Verlag.

3. A. Hanna, H. Ling, J. Furlong, Z. Yang, and M. Debbabi. Targeting Security Vulnerabilities: From Speci cation to Detection. In the Proceedings of the Quality Software, International Conference (QSIC 2008), August 12-13, 2008, Oxford, England, Pages 97-102, IEEE Press.

4. Z. Yang, A. Hanna, and M. Debbabi. Team Edit Automata for Testing Security Properties. In the Proceedings of the 3rd International Symposium on Information Assurance and Security (IAS'2007), August 29-31, 2007, Manchester, United Kingdom, IEEE Press.

5. N. Belblidia, M. Debbabi, A. Hanna, and Z. Yang. AOP Extension for FOSS Security Testing. In the Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering, CCECE'2006, May 7-10, 2006, Ottawa, Ontario, Canada, IEEE Press.

6. Laverdière, A. Mourad, A. Hanna, and M. Debbabi. Security Design Patterns: Survey and Evaluation. In the Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering, CCECE'2006, May 7-10, 2006, Ottawa, Ontario, Canada, Pages 1605-1608, IEEE Press.

Bibliography

[1] AT&T WinVNC Client/Server Buffer Overflow Vulnerabilities. Available at http://tools.cisco.com/security/center/viewAlert.x?alertId=1753. Last accessed timestamp: 2010.11.16.

[2] Autoconf. Available at http://www.gnu.org/software/autoconf/. Last access timestamp: 2010.11.23.

[3] Confidentiality Models. Available at http://www.blacksheepnetworks.com/security/info/misc/handbook/021-023.html. Last access timestamp: 2010.03.07.

[4] SecurityFocus. Available at http://www.securityfocus.com/bid/27796. Last access timestamp: 2011.07.06.

[5] A. A. Sofokleous and A. S. Andreou. Automatic, Evolutionary Test Data Generation for Dynamic Software Testing. In *The Journal of Systems and Software, Volume 81, Issue: 11, Pages 1883-1898*. Elsevier Science Inc., New York, NY, USA, November 2008.

[6] A. Chakrabarti and P. Godefroid. Software Partitioning for Effective Automated Unit Testing. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software EMSOFT '06, Seoul, Korea, Pages 262-271*. ACM, New York, NY, USA, October 2006.

[7] A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri. NUSMV: A New Symbolic Model Checker. In *International Journal on Software Tools for Technology Transfer, Volume 2*, 2000.

[8] A. Groce, D. Peled and M. Yannakakis. Adaptive Model Checking. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS '02, Pages 357-370*. Springer-Verlag, London, UK, 2002.

[9] A. Hanna, H. Z.u Ling, J. Furlong, and M. Debbabi. Towards Automation of Testing High-Level Security Properties. In *Proceeedings of the 22nd annual IFIP WG 11.3 working conference on Data and Applications Security, London, UK, July 13 2008, Pages 268-282*. Springer-Verlag.

[10] Á Hajnal and I. Forgács. An Applicable Test Data Generation Algorithm for Domain Errors. In *The ACM Special Interest Group on Software Engineering, Volume 23, Issue: 2, Pages 63-72*. ACM, New York, NY, USA, March 1998.

[11] J. H. Allen, S. Barnum, R. J. Ellison, G. McGraw, and N. R. Mead. In *Software Security Engineering: A Guide for Project Managers*. Addison-Wesley, ISBN: 978-0321509178, May 2008.

[12] B. Alpern and F. Schneider. Recognizing safety and liveness. In *Distributed Computing, Volume 2, Pages 117-126*. Springer Berlin / Heidelberg, 1987.

[13] B. Alwis. Apostle: Aspect Programming for Smalltalk. Available at `http://www.cs.ubc.ca/labs/spl/projects/apostle`. Last access timestamp: 2011.12.08.

[14] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *Formal Methods Europe (FME 02). Lecture Notes in Computer Science, Volume. 2391, Pages 21-40*. Springer, Berlin, Germany, 2002.

[15] BBC News. Microsoft to Patch 17-Year-Old Computer Bug. February 2010. Available at `http://news.bbc.co.uk/2/hi/8499859.stm`. Last access timestamp: 2011.11.14.

[16] N. Belblidia, M. Debbabi, A. Hanna, and Z. Yang. AOP Extension for Security Testing of Programs. In *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE-CCGEI), Ottawa, Canada Pages 647-650*. IEEE, June 2006.

[17] Berlecon Research GmbH. Free/Libre Open Source Software: Survey and Study. International Institute of Infonomics, University of Maastricht and Berlecon Research GmbH, July 2002. Available at `http://www.berlecon.de/studien/downloads/200207FLOSS_Activities.pdf`. Last access timestamp: 2011.07.06.

[18] D. Bird and C. Munoz. Automatic Generation of Random Self-checking Test Cases. In *IBM Systems Journal, Volume 22 No. 3, Pages 229-245*, 1983.

[19] D. Blankenhorn. The War is Over and Linux Won. November 2006. Available at `http://blogs.zdnet.com/open-source/?p=837`. Last access timestamp: 2011.06.07.

[20] R. Boyer, B. Elspas, and K. Levitt. SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software, Volume 10, No. 6, Pages 234-245*. ACM, 1975.

[21] L. Brim, I. Černá, P. Vařeková, and B. Zimmerova. Component-interaction Automata as a Verification-Oriented Component-Based System Specification. In *Proceedings of the 2005 Conference on Specification and Verification of Component-Based Systems, Volume 31, No. 2, Article 4*. ACM, 2005.

[22] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining Test Case Generation and Runtime Verification. In *Theoretical Computer Science, Volume 336, Issue: 2-3, Pages 209-234*. Elsevier Science Publishers Ltd., Essex, UK, May 2005.

[23] C. B. Jones. *Systematic Software Development using VDM, 2nd edition.* Prentice Hall International Science in Computer Science, Prentice Hall, NJ. ISBN: 0-13-880733-7, 1991.

[24] C. G. Broyden. A Class of Methods for Solving Nonlinear Simultaneous Equations. In *Mathematics of Computation, Volume 19, Pages 577-593*. American Mathematical Society, October 1965.

[25] C. Meudec. Automatic Generation of Software Tests from Formal Specifications. In *Ph.D. dissertation. Queen's University of Belfast, Northern Ireland, U.K.*, March 1998.

[26] C.-S. D.Yang, A. L. Souter and L. L. Pollock. All-Du-Path Coverage for Parallel Programs. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis ISSTA '98, Clearwater Beach, Florida, United States, Pages 153-162*. ACM, New York, NY, USA, 1998.

[27] C. Yan and W. Dan. A Scenario Driven Approach for Security Policy Testing Based on Model Checking. In *International Conference on Information Engineering and Computer Science ICIECS 200, Pages 1-4* , December 2009.

[28] C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of the 12th SPIN Workshop on Model Checking Software, San Francisco, USA, August 22-24, 2005 Pages 2-23*. Springer Verlag.

[29] S. Callanan, R. Grosu, H. Xiaowan, S. Smolka, and E. Zadok. Compiler-Assisted Software Verification using Plug-ins. In *Proceedings of the Next Generation Software Workshop (NGS '06) at IPDPS, Pages 8-15*. IEEE Press, 2006.

[30] M. Chakraborty and U. Chakraborty. An Analysis of Linear Ranking and Binary Tournament Selection in Genetic Algorithms. In *International Conference on Information, Communications and Signal Processing*. ICICS, September 1997.

[31] R. N. Charette. Why Software Fails. *IEEE Spectrum*, September 2005. Available at `http://spectrum.ieee.org/computing/software/why-software-fails`. Last access timestamp: 2011.06.07.

[32] Cigital and National Science Foundation. Genetic Algorithms for Software Test Data Generation. Available at `http://www.cigital.com/labs/projects/1008/`. Last access timestamp: 2011.07.06.

[33] Cisco Systems Inc. Cisco Security Advisory: $7xx$ Router Password Buffer Overflow. June 1998. Available at `http://www.cisco.com/warp/public/707/cisco-sa-19971216-pw-buffer.shtml`. Last access timestamp: 2011.07.06.

[34] L. Clarke. A system to generate test data and symbolically execute programs. In *IEEE Transactions on Software Engineering, Volume 2, No. 3, Pages 215-222*. IEEE Press, 1976.

[35] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the Modularity of Path-specific Customization in Operating System Code. In *In Proceedings of Foundations of software Engineering, Vienne, Austria.*, 2001.

[36] D. Beyer, T. A. Henzinger, R. Jhala and R. Majumdar. The Software Model Checker BLAST: Applications to Software Engineering. In *The International Journal on Software Tools for Technology Transfer (STTT), Volume 9, Issue: 5, Pages 505-525*. Springer-Verlag, Berlin, Heidelberg, October 2007.

[37] D. Detlefs, G. Nelson and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. In *ACM Journal, Volume 52, Isuue: 3, Pages 365-473*. ACM, New York, NY, USA, May 2005.

[38] D. Harel and E. Gery. Executable Object Modeling with Statecharts. In *IEEE Computer, Volumne 30, Pages 31-42*, 1997.

[39] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite-State Machines. In *Proceedings of the IEEE, Volume: 84, Issue: 8, Pages 1090-1123*. IEEE, August 1996.

[40] D. Peled. Model Checking and Testing Combined. In *Proceedings of the 30th International Conference on Automata, Languages and Programming ICALP '03, Pages 47-63*. Springer-Verlag, Berlin, Heidelberg, 2003.

[41] D. Peled, M. Y. Vardi and M. Yannakakis. Black box checking. In *The Journal of Automata, Languages and Combinatorics, Volume 7, Issue: 2, Pages 225-246*. Otto-von-Guericke-Universitat, Magdeburg, Germany, Germany, November 2001.

[42] D. Y. W. Park, U. Stern, J. U. Skakkebk and D. L. Dill. Java Model Checking. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering ASE'00*. IEEE Computer Society, Washington, DC, USA, 2000.

[43] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, and W. Wenhua. Detecting Vulnerabilities in C Programs using Trace-Based Testing. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, July 2010.

[44] J. Daemen and V. Rijmen. AES Proposal: Rijndael. Spring 1997. Available at `http://csrc.nist.gov/archive/aes/rijndael/misc/nissc2.pdf`. Last access timestamp: 2011.07.06.

[45] Department of Homeland Security, USA. Build Security In - Coding Rules, April 2007. Available at `https://buildsecurityin.us-cert.gov/daisy/bsi-rules/home/g1/848-BSI.html`. Last access timestamp: 2009.04.08.

[46] E. Farchi, A. Hartman and S. S. Pinter. Using a Model-Based Test Generator to Test for Standard Conformance. In *IBM Systems Journal, Volume 41, Issue: 1, Pages 89-110*. IBM Corp., January 2002.

[47] E. Gunter and D. Peled. Model Checking, Testing and Verification Working Together. In *The Journal of Formal Aspects of Computing, Volume 17, Issue:2, Pages 201-221*. Springer-Verlag, London, UK, UK, August 2005.

[48] E. M. Clarke, E. A. Emerson and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. In *ACM Transactions on Programming Languages and Systems, Volume 8, Pages 244-263*, 1986.

[49] E. M. Clarke, E. A. Emerson and J. Sifakis. Model Checking: Algorithmic Verification and Debugging. In *Communications of the ACM, Volume 52, Issue: 11, Pages 74-84*. ACM, New York, NY, USA, November 2009.

[50] E. M. Clarke, O. Grumberg and D. A. Peled. *Model Checking.* MIT Press ISBN: 978-0262032704, January 2000.

[51] D. Ebner. GIMPLE to XML patch. Available at `http://www.complang.tuwien.ac.at/cd/ebner/`. Last access timestamp: 2011.01.23.

[52] Electronic Privacy Information Systems. The Clipper Chip. *http://epic.org/crypto/clipper/*. Available at `http://epic.org/crypto/clipper/`. Last access timestamp: 2011.07.09.

[53] C. Ellis. Team Automata for Groupware Systems. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work, Pages 415-424, NY, USA*. ACM, 1997.

[54] R. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation. In *ACM Transactions on Software Engineering and Methodology, Volume 5, Pages 63-86*. ACM, January 1996.

[55] Free Software Foundation Inc. GCC, the GNU Compiler Collection. Available at `http://gcc.gnu.org/`. Last access timestamp: 2011.05.11.

[56] Free Software Foundation Inc. GNU Make. Available at `http://www.gnu.org/software/make/`. Last access timestamp: 2011.06.07.

[57] Free Software Foundation Inc. *The GCC Internals*, 1.3 edition, 2010. Available at `http://gcc.gnu.org/onlinedocs/gccint/`. Last access timestamp: 2010.11.09.

[58] S. Frei and T. Kristensen. The Security Exposure of Software Pottfolios. February 2010. Available at `http://secunia.com/gfx/pdf/Secunia_RSA_`

Software_Portfolio_Security_Exposure.pdf. Last access timestamp: 2011.02.21.

[59] G. Fraser, F. Wotawa and P. Ammann. Issues in Using Model Checkers for Test Case Generation. In *The Journal of Systems and Software, Volume 82, Issue: 9, Pages 1403-1418*. Elsevier Science Inc., New York, NY, USA, September 2009.

[60] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual, First Edition, ISBN: 0-321-22862-6*. Addison-Wesley Professional, 2003.

[61] J. Galarowics and B. Mohr. Analyzing Message Passing Programs on the Cray T3E with PAT and VAMPIR. *Technical report, ZAM Forschungszentrum: Juelich, Germany*, 1998.

[62] L. M. Garshol. BNF and EBNF: What Are They and How do They Work? Available at http://www.garshol.priv.no/download/text/bnf.html. Last access timestamp: 2007.09.04.

[63] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Pages 213-223*. ACM, June 2005.

[64] Google Inc. The Go Programmming Language. 2007. Available at http://code.google.com/p/go. Last access timestamp: 2011.08.11.

[65] S. Guyer, E. Berger, and C. Lin. Detecting Errors with Configurable Wholeprogram Dataflow Analysis. PLDI 2002 Berlin, Germany, 2002. Available at http:

//www.cs.utexas.edu/~lin/papers/detecting02.pdf. Last access timestamp: 2011.10.03.

[66] H. Haralambiev, S. Boychev, D. Lilov and K. Kraichev. Applying Source Code Analysis Techniques: A Sase Study for a Large Mission-Critical Software System. In *IEEE International Conference on Computer as a Tool (EUROCON), 2011, Page 1*, April 2011.

[67] H. Shahriar and M. Zulkernine. Automatic Testing of Program Security Vulnerabilities. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference COMPSAC '09, Volume 2, Pages 550-555*. IEEE Computer Society, Washington, DC, USA, 2009.

[68] H. Song, W. Liang, Z. Changyou and Y. Hong. A Software Security Testing Method Based on Typical Defects. In *International Conference on Computer Application and System Modeling ICCASM, 2010, Volume 5, Pages 150-153*, October 2010.

[69] H. Z. Ling. Towards The Automation of Vulnerability Detection in Source Code. Master's Thesis, Concordia University, Montreal, Canada, 2009.

[70] A. Hanna, , H. Z. Ling, and M. Debbabi. Dynamic detection of security vulnerabilities through team edit automata and the security chaining approach. In *Proceedings of the Eighth International Conference on Quality Software (QSIC'08), Pages 97-102, Oxford, UK*. IEEE Computer Society Press, August 2008.

[71] A. Hanna, H. Z. Ling, X. Yang, and M. Debbabi. A synergy between static and dynamic analysis for the detection of software security vulnerabilities. In *The 4th*

International Symposium on Information Security (IS'09), Page 815, Vilamoura, Algarve-Portugal. Springer-Verlag, November 2009.

[72] I. Aktug and K. Naliuka. ConSpec – A Formal Language for Policy Specification. In *Electronic Notes in Theoretical Computer Science, Volume 197, Issue: 1, Pages 45-58*. Elsevier Science Publishers B. V., Amsterdam, Netherlands, February 2008.

[73] I. Lynce and J. Marques-Silva. Building State-Of-The-Art SAT Solvers. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence ECAI 2002, Pages 166-170*, 2002.

[74] M. Ivaldi. Linux 2.6.x suid_dumpable vulnerability. July 2006. Available at `http://www.exploit-db.com/exploits/2031/`. Last access timestamp: 2011.07.06.

[75] J. C. Corbett, M. B. Dwyer, J. Hatcliff and Robby. Bandera: a Source-Level Interface for Model Checking Java Programs. In *Proceedings of the 22nd International Conference on Software Engineering ICSE '00, Limerick, Ireland, Pages 762-765*. ACM, New York, NY, USA, 2000.

[76] J.-C. Lin, and P.-L. Yeh. Automatic test data generation for path testing using GAs. January 2001.

[77] J. Callahan, F. Schneider and S. Easterbrook. Automated Software Testing Using Model-Checking. In *Proceedings 1996 SPIN Workshop, Pages 118-127*. Rutgers University, Brunswick, NJ, August 1996.

[78] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering, Linköping, Pages 21-28*. ECSEL, October 1999.

[79] J. M. Spivey. *The Z Notation: A Reference Manual, 2nd edition.* Prentice Hall International Science in Computer Science, Prentice Hall, NJ. ISBN: 0-139-78529-9, June 1992.

[80] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge University Press, Cambridge, U.K. ISBN: 978-0521054140, February 2008.

[81] J. Miller, M. Reformat and H. Zhang. Automatic Test Data Generation Using Genetic Algorithm and Program Dependence Graphs. In *Information and Software Technology, Volume 48, Pages 586-605*, 2006.

[82] J. R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, Cambridge, U.K. ISBN: 978-0521496193, October 1996.

[83] J. Romero-Mariona, H. Ziv and D.J. Richardson. Increasing Trustworthiness through Security Testing Support. In *IEEE Second International Conference on Social Computing (SocialCom), 2010, Pages 920-925*, August 2010.

[84] Y. Jia, Z. Li, and Z. Zhang. Timed Component-Interaction Automata for Specification and Verification of Real-Time Reactive Systems. In *International Conference on Computer Science and Software Engineering, Volume 2, Pages 135-138*. IEEE Computer Society, 2008.

[85] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, Cambridge, MA. ISBN: 978-0262133418, February 1998.

[86] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Overview of AspectJ. In *In Proceedings of the 15th European Conference (ECOOP'01), Pages 327-353, Budapest, Hungary*. Springer Verlag, 2001.

[87] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming (ECOOP), Jyväskylä, Finland, June 9-13m 1997, Pages 220-242*. Springer-Verlag.

[88] S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped - A Model-Checker for Pushdown Systems. Available at `http://www.fmi.uni-stuttgart.de/szs/tools/moped/`. Last access timestamp: 2011.07.06.

[89] H. Kim. An AOSD Implementation for C#. Technical Report TCD-CS2002-55, Department of Computer Science, Trinity College, Dublin, 2002.

[90] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Computational Analysis of Run-time Monitoring: Fundamentals of Java-MaC. In *Electronic Notes in Theoretical Computer Science, Volume 70, No. 4, Pages 1-15*. Elsevier B.V., December 2002.

[91] B. Korel. A Dynamic Approach of Test Data Generation. In *Proceedings on Software Maintenance, San Diego, USA, November 26-29, 1990, Pages 311-317*. IEEE, 1990.

[92] B. Korel. Automated software test data generation. *IEEE Transactions on Software Enfineering, Vol. 16 No 8.*, August 1990.

[93] D. kranzlmueller. Event Graph Analysis for Debugging Massively Parallel Programs. September 2000. Available at `http://www.mcs.anl.gov/~norris/MPIGraphAnalysis-KranzlMueller.pdf`. Last access timestamp: 2011.07.06.

[94] L. Bauer, J. Ligatti and D. Walker. A Language and System for Composing Security Policies. Technical Report TR-681-03, Princeton University, 2004.

[95] L. Bauer, J. Ligatti and D. Walker. Composing Security Policies with Polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '05, Chicago, IL, USA, Pages 305-314*. ACM, New York, NY, USA, 2005.

[96] L. Gallagher, J. Offutt and A. Cincotta. Integration Testing of Object-Oriented Components using Finite State Machines: Research Articles. In *Software Testing, Verification And Reliability, Volume 16, Issue: 4, Pages 215-266*. John Wiley and Sons Ltd., Chichester, UK, December 2006.

[97] J. Larus. EEL: An Executable Editing Library. 1996. Available at `http://pages.cs.wisc.edu/~larus/eel.html`. Last access timestamp: 2011.07.06.

[98] M.-A. Laverdière, A. Mourad, A. Hanna, and M. Debbabi. Security design patterns: Survey and evaluation. In *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering, Ottawa, Canada, May 22, 2006, Pages 1605-1608*. IEEE, 2006.

[99] S. Liang. Statistical Sampling vs. Code Instrumentation. 1998. Available at `http://www.usenix.org/events/coots99/full_papers/liang/liang_html/node9.html`. Last access timestamp: 2011.07.06.

[100] K. J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, 1st Edition, August 28, 1995, ISBN: 978-0534946029, PWS Publishing Company, Boston. 1996.

[101] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. In *International Journal of Information Security, Volume 4, No. 1-2, Pages 2-16*, December 2005.

[102] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. In *CWI Quarterly Volume 2, Pages 1-30*. Laboratory for Computer Science, Massachusetts Institute of Technology, September 1989.

[103] M.-C. Gaudel. Testing can be formal too. In *Theory and Practice of Software Development TAPSOFT '95: 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, Pages 82-96*. Springer-Verlag, Berlin, Germany, May 1995.

[104] M. Gallagher and V. L. Narasimhan. ADTEST: A Test Data Generation Suite for Ada Software Systems. In *IEEE Transactions on Software Engineering, Volume 23, Issue: 8, Pages 473-484*. IEEE Press, Piscataway, NJ, USA, August 1997.

[105] M. Holcombe and F. Ipate. *Correct Systems - Building a Business Process Solution.* Applied computing. Springer, 1998.

[106] M. L. Soffa, A. P. Mathur and N. Gupta. Generating Test Data for Branch Coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ASE '00, Page 219*. IEEE Computer Society, Washington, DC, USA, 2000.

[107] M. Stephan. Model Checking: A Tutorial Overview. In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes MOVEP, London, UK, Pages 3-38*. Springer-Verlag, 2001.

[108] M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the Symposium on Logic in Computer Science (LICS), Pages 332-344*. IEEE Computer Society Press, Los Alamitos, CA, 1986.

[109] M. Vaziri-Farahani. Finding Bugs in Software with a Constraint Solver. In *Ph.D. dissertation. Massachusetts Institute of Technology, USA.*, February 2004.

[110] Michael Richmond. Finding Roots by "Open" Methods. Available at `http://spiff.rit.edu/classes/phys317/lectures/open_root/open_root.html#difference`. Last access timestamp: 2011.10.22.

[111] Microsoft Corporation. Microsoft Security Bulletin MS04-028. Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution. December 2004. Available at `http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx`. Last access timestamp: 2011.03.07.

[112] Microsoft Corporation. How to use security zones in Internet Explorer. January 2007. Available at `http://windows.microsoft.com/en-CA/windows-vista/Change-Internet-Explorer-Security-settings`. Last access timestamp: 2011.07.06.

[113] R. Milner. A Theory of Type Polymorphism in Programming. In *Journal of Computer and System Sciences, Volume 17, No. 3, Pages 348-375*, 1978.

[114] M. Minoux. Mathematical programming theory and algorithm. *Wiley-Interscience Publication*, 1986.

[115] A. Mourad, M.-A. Laverdière, and M. Debbabi. Security Hardening of Open Source Software. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST) 2006, Markham, Ontario, Canada, October 30 - November 1, 2006, Volume 380*. ACM.

[116] N. Gupta, A. P. Mathur and M. L. Soffa. UNA Based Iterative Test Data Generation and its Evaluation. In *Proceedings of the 14th IEEE international conference on Automated software engineering ASE '99, Page 224*. IEEE Computer Society, Washington, DC, USA, 1999.

[117] N. Mansour and M. Salame. Data Generation for Path Testing. *Software Quality Control Journal, Volume 12, Issue: 2, Pages 121-136*, June 2004.

[118] N. Shahmehri, A. Mammar, E. Montes de Oca, D. Byers, A. Cavalli, S. Ardi and W. Jimenez. An Advanced Approach for Modeling and Detecting Software Vulnerabilities. In *Information and Software Technology, Volume 54, Issue: 9, Pages 997-1013*. Butterworth-Heinemann, Newton, MA, USA, September 2012.

[119] N. T, Sy and Y. Deville. Automatic Test Data Generation for Programs with Integer and Float Variables. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering ASE '01, Page 13*. IEEE Computer Societym, Washington, DC, USA, 2001.

[120] N. Tracey, J. Clark, K. Mander and J. McDermid. An Automated Framework for Structural Test-Data Generation. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering ASE '98, Page 285*. IEEE Computer Society, Washington, DC, USA, 1998.

[121] Netcraft. Web Server Survey. May 2007. Available at `http://news.netcraft.com/archives/category/web-server-survey/`. Last access timestamp: 2011.07.09.

[122] D. Novillo. Tree SSA: A New Optimization Infrastructure for GCC. In *Proceedings of the GCC Developers Summit, Pages 181-193*, 2003.

[123] Numerical Methods Guys, University of North Florida - Online Lecture Notes. Backward Divided Difference: Numerical Differentiation. Available at `http:`

//www.youtube.com/watch?v=CEouVWYeKrk. Last access timestamp: 2011.03.08.

[124] Numerical Methods Guys, University of North Florida - Online Lecture Notes. Newton-Raphson Method: Advantages and Drawbacks. Available at http://www.youtube.com/watch?v=QwyjgmqbR9s&feature=relmfu. Last access timestamp: 2011.03.08.

[125] P. M. S. Bueno and M. Jino. Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering ASE '00, Page 209*. IEEE Computer Society, Washington, DC, USA, 2000.

[126] P. McMinn, M. Harman, D. Binkley and P. Tonella. The Species per Path Approach to Search Based Test Data Generation. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis ISSTA '06, Portland, Maine, USA, Pages 13-24*. ACM, New York, NY, USA, 2006.

[127] Patrick H. Worley. MPICL, Portable Instrumentation Library. 2000. Available at http://www.csm.ornl.gov/picl/mpicl.html. Last access timestamp: 2011.07.09.

[128] PGP Corporation/Symantec. Pretty Good Privacy. Available at http://www.symantec.com/business/theme.jsp?themeid=pgp. Last access timestamp: 2011.07.09.

[129] R. Charpentier, M. Debbabi, A. Hanna, and TFOSS Team. Security Evaluation and Hardening of Free and Open Source (FOSS). *Electronic Communication of the European Association of Software Science and Technology (ECEASST), Volume 33*, September 2010.

[130] R. Dharam, and S. G. Shiva. A Framework for Development of Runtime Monitors. In *International Conference on Computer Information Science (ICCIS), 2012, Volume 2, Pages 953-957*, June 2012.

[131] R. Hassan, M. Eltoweissy, S. Bohner and S. El-Kassas. Formal Analysis and Design for Engineering Security Automated Derivation of Formal Software Security Specifications from Goal-Oriented Security Requirements. In *IET Software, Volume 4, Issue: 2, Pages 149-160*, April 2010.

[132] R. P. Pargas, M. J. Harrold and R. R. Peck. Test-Data Generation Using Genetic Algorithms. In *Software Testing, Verification And Reliability, Volume 9, Issue: 9, Pages 263-282*, 1999.

[133] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. In *Proceedings of the 43rd IEEE Conference on Decision and Control (CDC), Volume 21, No. 2, Pages 120-126*. ACM, February 1978.

[134] R.M. Hierons. Testing from a Z specification. In *Software Testing, Verification and Reliability, Volume 7, Issue: 1, Pages 19-33*. John Wiley & Sons, Ltd., March 1997.

[135] R.M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward and H. Zedan. Using Formal Specifications to Support Testing. In *The ACM Computing Surveys (CSUR) Journal, Volume 41, Issue: 2, Article 9, Pages 1-76*. ACM, New York, NY, USA, February 2009.

[136] M. Ronsse and K. D. Bosschere. JiTI: Tracing Memory References for Data Race Detection. In *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, Bonn, Germany*, 1997.

[137] RSA Laboratories. The RSA Factoring Challenge. 2009. Available at `http://www.rsa.com/rsalabs/node.asp?id=3723`. Last access timestamp: 2011.07.06.

[138] S. Burton. Automated Testing from Specifications. In *Ph.D. dissertation. The University of York, York, U.K.*, March 2002.

[139] S.-D. Gouraud, A. Denise, M.-C.Gaudel and B. Marre. A New Way of Automating Statistical Testing Methods. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering ASE '01, Page 5*. IEEE Computer Society, Washington, DC, USA, 2001.

[140] S. H. Edwards. A Framework for Practical, Automated Black-Box Testing of Component-Based Software. In *Software Testing, Verification And Reliability, Volume 11, Page 2001*, 2001.

[141] S. Spacey, A. Simon, W. Wiesemann, D. Kuhn and W. Luk. Robust Software Partitioning with Multiple Instantiation. In *INFORMS Journal on Computing, Volume 24, Issue: 3, Pages 500-515*. INFORMS, July 2012.

[142] F. B. Schneider. Enforceable Security Policies. In *ACM Transactions on Information and System Security (TISSEC), Volume 3, No. 1, Pages 30-50*. ACM, February 2000.

[143] R. C. Seacord. Secure Coding in C and C++. *SEI Series. Addison-Wesley. ISBN: 978-0321335722*, 2005.

[144] Security Tracker. Netpbm Uses Unsafe Temporary Files and May Let Local Users Gain Elevated Privileges. January 2004. Available at `http://www.securitytracker.com/alerts/2004/Jan/1008757.html`. Last access timestamp: 2011.07.09.

[145] Simes. How to Break Out of a chroot Jail. 2002. Available at `http://www.bpfh.net/simes/computing/chroot-break.html`. Last access timestamp: 2011.09.24.

[146] H. Sneed. Estimating the Costs of a Reengineering Project. In *Proceedings of the* 12*th Working Conference on Reverse Engineering (WCREŠ05), Pittsburgh, USA, November 11, 2005, Page 119*, 2005.

[147] O. Spinczyk, A. Gal, and W. chroder Preikschat. AspectC++: An aspect-oriented Extension to C++. In *In Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (CRPIT'02), Sydney, Australia, Pages 53-60*. Australian Computer Society, 2002.

[148] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. A Technical Report. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), Volume 29, No. 6, Pages 196-205, Palo Alto, California, USA, June 1994*. ACM.

[149] T. Ball, B. Cook, V. Levin and S. K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods Inside Microsoft. In *Integrated Formal Methods, 4th International Conference IFM 2004, Canterbury, UK, Pages 1-20*. Springer, April 2004.

[150] T. Ball, V. Levin and S. K. Rajamani. A Decade of Software Model Checking with SLAM. In *Communications of the ACM, Volume 54, Issue: 7, Pages 68-76*. ACM, New York, NY, USA, July 2011.

[151] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement for limited-memory systems. In *Proceedings of the International Conference on Privacy, Security and Trust(PST'06), Article No. 38, Pages 1-12, Markham, Ontario, Canada*. ACM, 2006.

[152] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. *N* Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software engineering, Pages 107-119*. ACM, 1999.

[153] T.S. Chow. Robust Software Partitioning with Multiple Instantiation. In *IEEE Transactions on Software Engineering, Volume SE-4, Issue: 3, Pages 178-187*. IEEE, May 1978.

[154] U. Erlingsson. The Inlined Reference Monitor Approach to Security Policy Enforcement. Ph.D. dissertation. Cornell University, Ithaca, NY, USA, 2004.

[155] US Code Collection. United States Code, Title 44, Chapter 35, Subchapter III. Available at `http://www.law.cornell.edu/uscode/44/3542.html`. Last access timestamp: 2009.10.20.

[156] US Department of Commerce/National Institute of Standards and Technology. Digital signature standard (DSS). January 2000. Available at `http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf`. Last access timestamp: 2009.10.02.

[157] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *the 16th Annual Computer Security Applications Conference (ACSAC 2000), Page 257, December 11-15 2000, New Orleans, Louisiana, USA*. IEEE Computer Society, 2000.

[158] W. Grieskamp, Y. Gurevich, W. Schulte and M. Veanes. Generating Finite State Machines from Abstract State Machines. In *Proceedings of International Symposium on Software Testing and Analysis ISSTA, Pages 112-122*, 2002.

[159] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software engineering (ICSE'81), Pages 439-449, Piscataway, NJ, USA*. IEEE Press, 1981.

[160] D. Wheeler. Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers! April 2007. Available at `http://www.dwheeler.com/oss_fs_why.html`. Last access timestamp: 2011.07.09.

[161] X. Liu, H. Liu, B. Wang, P. Chen and X. Cai. A Unified Fitness Function Calculation Rule for Flag Conditions to Improve Evolutionary Testing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering ASE '05, Long Beach, CA, USA, Pages 337-341*. ACM, New York, NY, USA, 2005.

[162] Z. Yang, A. Hanna, and M. Debbabi. Team Edit Automata for Testing Security Property. In *Proceedings of the Third International Symposium on Information Assurance and Security (IAS '07), August 29-31, 2007, Manchester, UK, Pages 235-240*. IEEE Computer Society, 2007.

[163] Z. Hui, S. Huang, B. Hu and Z. Ren. A taxonomy of software security defects for SST. In *International Conference on Intelligent Computing and Integrated Systems ICISS, 2010, Pages 99-103*, October 2010.

Appendix A

Compiler Implementation for Instrumentation Guide Language

This appendix presents the grammar of the instrumentation guide language used by our GCC extension. Below is the grammar specified in EBNF [62] form.

Grammar of the Instrumentation Guide Language

```
/*------------------------------------------
 * PARSER RULES
 *------------------------------------------*/
prog := stat+ ;

stat
    := statFuncCall ENDOFSTMT
    |  statVarUse ENDOFSTMT
    |  statVarDecl ENDOFSTMT
    |  statOutOfScope ENDOFSTMT
    ;

statFuncCall
    := 'inscope' scopeId flexibleInstrPoint
       'callfunc' LPAREN normalId RPAREN
       'inject' '"' ID '"' exposeArgs exposeReturn
    ;
```

```
statVarUse
    := flexibleInstrPoint varUseOptions LPAREN varIdString RPAREN
       'inject' '"' ID '"'
    ;

statVarDecl
    := 'after' 'declvar' LPAREN varIdString RPAREN
       'inject' '"' ID '"'
    ;

statOutOfScope
    := 'before' 'exitfunc' LPAREN normalId RPAREN
       'inject' '"' ID '"'
    |  'before' 'exitscope' LPAREN varIdString RPAREN
       'inject' '"' ID '"'
    ;

scopeId
    := '"' ID '"'
    |  WILDCHAR
    ;

flexibleInstrPoint
    := 'before'
    |  'after'
    ;

normalId
    := '"' ID '"'
    |  WILDCHAR
    ;

exposeArgs
    := 'exposeargs'
    |  'noexposeargs'
    ;

exposeReturn
    := 'exposereturn'
    |  'noexposereturn'
    ;

varUseOptions
    := 'readvar'
    |  'writevar'
    |  'derefptr'
    ;

varIdString
    := fileNameString DBLCOLON LINENUM DBLCOLON varId
    ;

varId
```

```
    := '"' e=ID '"'
    |  WILDCHAR
    ;

fileNameString
    := '"' FILENAME '"'
    |  WILDCHAR
    ;

/*----------------------------------------
 * LEXER RULES
 *----------------------------------------*/
FILENAME := ('0'..'9'|'a'..'z'|'A'..'Z'|'_'|'.')+ '.' ('c'|'C');
ID        := ('a'..'z'|'A'..'Z'|'_')
             ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
DEREFID  := ('a'..'z'|'A'..'Z'|'_')
             ('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'.')* ;
DBLCOLON := '::';
ENDOFSTMT:= ';';
WS        := (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;} ;
LPAREN   := '(';
RPAREN   := ')';
WILDCHAR := '*';
LINENUM  := ('0'..'9')+;
```