

ON BUILDING A DYNAMIC SECURITY
VULNERABILITY DETECTION SYSTEM USING
PROGRAM MONITORING TECHNIQUE

ZHENRONG YANG

A THESIS

IN

THE CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE (INFORMATION SYSTEMS SECURITY)

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

APRIL 2008

© ZHENRONG YANG, 2008



Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN: 978-0-494-40905-3
Our file *Notre référence*
ISBN: 978-0-494-40905-3

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

Abstract

On Building a Dynamic Security Vulnerability Detection System Using Program Monitoring Technique

Zhenrong Yang

This thesis presents a dynamic security vulnerability detection framework that sets up an infrastructure for automatic security testing of Free and Open Source Software (FOSS) projects. It makes three contributions to the design and implementation of a dynamic vulnerability detection system. Firstly, a mathematical model called *Team Edit Automata* is defined and implemented for security property specification. Secondly, an automatic code instrumentation tool is designed and implemented by extending the GNU Compiler Collection (GCC). The extension facilitates seamless integration of code instrumentation into FOSS projects' existing build system. Thirdly, a dynamic vulnerability detection system is prototyped to integrate the aforementioned two techniques. Experiments with the system are elaborated to automatically build, execute, and detect vulnerabilities of FOSS projects. Overall, this research demonstrates that monitoring program with *Team Edit Automata* can effectively detect security property violation.

Acknowledgments

I would like to thank Dr. Mourad Debbabi, my advisor, for his continuous support and invaluable guidance throughout my academic program. He welcomed me to the Testing Free and Open Source Software (TFOSS) research team in late 2005. He gave me the freedom to pursue independent ideas, while challenging me with "why, what, how" from time to time to help me clarify my mind.

My appreciation extends to the members on my graduation committee: Dr. Joey Paquet, and Dr. Amr Youssef. Their valuable suggestions helped enhance the content of this thesis.

My sincere thanks to my fellow researchers: Dima Al-Hadidi, Mourad Azzam, Nadia Belblidia, Aiman Hanna, Marc-André Laverdière, Syrine Tlili, and Xiaochun Yang. Thanks to Rachid Hadjidj for helping me with the GUI implementation of the security property editor.

Finally and most importantly, my deepest appreciation to my wife Weili, my parents back in China, and my friends all over the world. Many thanks to them for their continuous support, understanding and love.

Contents

List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	4
1.3 Contributions	5
1.4 Organization of the Thesis	6
2 Detection of Security Vulnerabilities in Source Code	8
2.1 Security Vulnerabilities in Source Code	8
2.2 Software Vulnerability Detection	10
2.2.1 Static Program Analysis Techniques	10
2.2.2 Dynamic Program Analysis Techniques	20
2.3 Program Monitoring and Instrumentation	26
2.3.1 Program Monitoring	26
2.3.2 Program Instrumentation	27

2.4	Summary	31
3	Security Property Specification	32
3.1	Introduction	32
3.1.1	Motivation	32
3.1.2	Related Work	34
3.1.3	Security Property vs. Security Policy	36
3.2	Team Edit Automata	36
3.2.1	Edit Automata	37
3.2.2	Team Automata	39
3.2.3	Team Edit Automata	41
3.3	Implementing Team Edit Automata	47
3.3.1	Design Overview	47
3.3.2	Implementing Component Automata	49
3.3.3	Implementing Team Automata	49
3.4	Summary	51
4	Code Instrumentation	53
4.1	Extending GCC for Code Instrumentation	53
4.2	Introduction to the Extension	55
4.2.1	Workflow Overview	55
4.2.2	Functionality	56
4.2.3	User Interface	59
4.3	GCC Internals	63

4.3.1	GCC Architecture and Compilation Phases	63
4.3.2	GENERIC and GIMPLE Languages	65
4.4	Design and Implementation	66
4.4.1	Adding Instrumentation Passes	66
4.4.2	Adding Command-line Option	70
4.4.3	Recognizing Environment Variables	71
4.4.4	Instrumentation Phase One - Scope-Wise Instrumentation	71
4.4.5	Instrumentation Phase Two - CFG-Based Instrumentation	72
4.4.6	Instrumentation Example - Supporting Suppression	75
4.5	Summary	77
5	Integrated System and Experiments	79
5.1	Integrated System	79
5.1.1	System Overview	79
5.1.2	System Configuration	80
5.1.3	Project Management	82
5.1.4	Vulnerability Report, View and Fix	85
5.1.5	Security Property Specification	86
5.1.6	System Implementation	94
5.2	Experiments with the Integrated System	95
5.2.1	Experiment Environment	95
5.2.2	Experiment 1: Checking Memory Management Vulnerabilities	96
5.2.3	Experiment 2: Scalability and Usability Test	100

5.3 Summary	103
6 Conclusion	104
Bibliography	106
A Compiler Implementation for Instrumentation Guide Language	119
B Rough GIMPLE Grammar	122
C Source Codes Used in Experiment 1	126
C.1 Security Property Specification for Memory Management Vulnerabilities . .	126
C.1.1 MemorySM.sm: State Machine for States of Program Memory . . .	126
C.2 Vulnerable C Programs Evaluated in Experiment 1	128
C.2.1 Program 1	128
C.2.2 Program 2	129
C.2.3 Program 3	129
C.2.4 Program 4	130
C.2.5 Program 5	131
C.2.6 Program 6	131

List of Figures

1	Example of Macro-Assisted Code Instrumentation	28
2	Collaboration Graph of Team Edit Automata Implementation Classes	48
3	Workflow of <code>TeamAutomata::Query()</code>	50
4	Workflow of the GCC Extension for Code Instrumentation	55
5	GCC Architecture and Compilation Phases	64
6	Sample Source Code and Corresponding GIMPLE Representation	67
7	Pass Information for Instrumentation Phase One	68
8	Registering Code Instrumentation Passes in <code>passes.c</code>	69
9	Enabling Command-Line Option for Code Instrumentation in <code>common.opt</code>	70
10	The <i>execute function</i> of Instrumentation Phase One	73
11	Instrumentation of Code Suppression	76
12	Integrated System - Overview	80
13	Integrated System - Preference Dialog	81
14	Integrated System - Configure System Settings through <i>File</i> Menu	82
15	Integrated System - Project Management through <i>Project</i> Menu	82
16	Integrated System - Project Creation Dialog	83

17	Integrated System - Newly Created Project in Project List	83
18	Integrated System - Dialog of Project Building Result	84
19	Integrated System - Dialog of Project Execution Configuration	85
20	Integrated System - List of Property Violations	86
21	Integrated System - Detailed Vulnerability View	87
22	Integrated System - Source Code Editor for Bug Fixing	87
23	Integrated System - Overview of Security Property Editor	88
24	Integrated System - Specifying State Name in Security Property Editor . . .	89
25	Integrated System - Defining Transition Event in Security Property Editor .	91
26	Integrated System - Defining Transition Guard in Security Property Editor .	91
27	Integrated System - Defining Transition Actions in Security Property Editor	92

List of Tables

1	Static Security Analysis Tools	12
2	Dynamic Security Analysis Tools	20
3	Fields in the Instrumentation Guide Input File	62
4	Instrumentation Points and Corresponding Phases	72
5	GIMPLE APIs Used for Instrumenting Suppression Code	75
6	Experiment Result of Checking Memory Management Vulnerabilities . . .	98
7	FOSS Projects Used in Scalability and Usability Experiment	101

Chapter 1

Introduction

1.1 Motivation

In recent years, the development of Free and Open Source Software (FOSS) [68] has gained much attention. More and more organizations, including government and military, have started to consider the deployment of FOSS applications for cost-efficiency reason. In 2006, Forrester Research [4] suggested to North American and European enterprises that "firms should consider open source options for mission-critical applications" [46]. A similar advice was given by the State of California in 2004, urging that "the state should more extensively consider the use of open source software" [66].

Along with this trend rises the need for assuring security of the FOSS programs, because the impact of software security breach is disastrous to deploying organizations. For example, the "Melissa" computer virus unleashed in 1999 caused more than \$80 million in damage by corrupting personal computers and computer networks in business and government [67]. In worse scenarios, the impact not only incurs economic loss, but endangers

national security. For instance, the electronic messages from the U.S. Department of Defence regarding the country's nuclear activities were intercepted by a 16-year-old hacker in 1999 [81]. What is more, the scope of computer attacks is continuously expanding, making security concerns more necessary. Let numbers tell the facts: In 2001 and 2002, the Computer Emergency Response Team (CERT) [82] reported 52,658 and 82,094 computer attack incidents respectively. In 2003, the total number of reported attacks boosted to 137,529 cases, exceeding the sum of those reported in the previous two years¹.

In order to protect software against malicious attacks, researchers attempted to incorporate security concerns in various phases of its life cycle. Some suggested security design patterns and more "secure" programming languages to improve security in software's design and implementation. Others have tried to assure software security after its deployment. Their approaches vary from dynamic security enforcement to software security patching. However, none of these methodologies provide enough security assurance when FOSS is concerned. Designing and implementing security in software is certainly the best countermeasure to malicious attacks, because it minimizes or eliminates the chance of introducing vulnerabilities in software. Nonetheless, it does not mean much to organizations deploying FOSS programs, because they usually do not involve in the design and implementation of FOSS. Enforcing security properties is an active countermeasure against computer attacks. It is especially useful after software is deployed. However, the overhead of having an enforcing mechanism executing in parallel with the deployed software so far prevents this methodology from being practically useful. Releasing security patches to remedy existing

¹Statistic data for 2004 and later years are not publicized from CERT web site.

vulnerabilities in software is a traditional means to enhance security in software. Yet, vulnerabilities might be exploited and damage made before patches are released. Hence, this is the last means that deploying organizations can resort to.

So, what is the ideal solution for organizations to deploy FOSS while assuring its security? The answer is software security analysis. In brief, the analysis detects security vulnerabilities in software through reasoning on its various attributes - structure, documentation, runtime behavior, etc. Ideally, vulnerabilities are detected before software is deployed. To achieve this goal, researchers have already devised many tools. The majority of these tools statically analyze the source code of software to detect vulnerabilities. Due to their efficiency and improving accuracy, static security analysis tools have risen to great prominence in the past few years. However, not all security vulnerabilities are statically decidable [94]. In addition, some analysis problems can be solved more efficiently through dynamic analysis than through static one. Consequently, our research is greatly motivated by the requirement for a dynamic security analysis solution, which can complement the weaknesses of static program analysis.

So far, no existing dynamic vulnerability detection tools provide satisfactory solution of checking system-specific security properties in software. Many tools only detect very specific security vulnerabilities, e.g. runtime memory errors. As for those that do perform general purpose dynamic analysis, only few ones offer explicit support to security vulnerability detection. For example, the JNuke [19] tool implements a program monitoring platform and defines a set of APIs for security analysts to interact with the monitoring system. Yet, it is security analysts' full responsibility to define how to check system-specific security properties using JNuke APIs. Other tools such as EXE [28] and CCured [65] rely

on runtime assertions to detect security property violation, constraining themselves from checking many temporal security properties.

We believe that there is a need to build a dynamic security vulnerability detection tool, which can be easily extended to check user-defined system-specific security properties. It facilitates the discovery of security vulnerabilities in software. When combined with an automatic test suite generator, it can be used to automatically detect vulnerabilities, some of which static analysis tools cannot detect accurately and/or efficiently.

1.2 Objectives

The research reported in this thesis sets out to build a dynamic security vulnerability detection system as an infrastructure for automatic or semi-automatic software security testing. The system shall allow security analysts to specify system-specific security properties. It shall also transparently convert the specifications to observable runtime behaviors, so that violations of user-defined security properties can be detected when the testee program is executed.

Accordingly, our research effort has the following objectives:

- Conduct an in-depth study of existing software security evaluation techniques.
- Define a mechanism for security property specification and design/implement a software utility to allow security analysts to write system-specific security properties.
- Design and implement a software utility to dynamically detect violations of user-specified security properties.

- Design and implement a software utility for automatic code instrumentation. The utility shall be able to instrument code at various security-sensitive program points defined by security analysts. It shall also seamlessly integrate in the existing build system so that deploying organizations can easily use our tool to evaluate FOSS projects.
- Experiment our detection framework on some FOSS projects to assess the viability and usability of our approach.

1.3 Contributions

This thesis makes the following contributions to the development of an automatic and extensible solution for software security testing.

- It proposes a framework for dynamically detecting system-specific security vulnerabilities in software. The framework can be used jointly with an automatic test suite generator to build automatic security testing tools. It minimizes the human effort that is needed for the security analysis such that the analysts only need to specify the tested security properties. The remaining operations of translating a property specification into checkable program behaviors and detecting property violations are transparent to the analysts. In addition, the framework is extensible such that analysts are free to specify any temporal security properties they intend to check. An infrastructure is accordingly designed and prototyped to support this framework.
- It defines a new mathematical model called *Team Edit Automata* to specify security

properties. The new model is used to describe temporal security properties and to capture the interaction among various components in testee programs. When multiple security properties define different reaction to identical program actions, the model allows for determining which property specification should be respected. The implementation of *Team Edit Automata* is prototyped as program monitors to observe the behavior of programs when they are executed.

- It elaborates a new program instrumentation solution through extending the GNU Compiler Collection (GCC) [8]. An extension of GCC is prototyped to allow automatic code instrumentation on C source code. This solution has two attractive properties. Firstly, it can be extended to support the programming languages whose front ends are contained in GCC. Currently, the main GCC distribution contains front ends for C, C++, Objective C, Fortran, Java, and Ada [6]. Accordingly, the solution can be extended to support all these languages. Secondly, it can be easily incorporated into automatic build systems, such as GNU Make [9]. Users only need to define two environment variables to enable the code instrumentation functionality. Such convenience is valuable to dynamic analysis of FOSS projects, because many of them use GCC as the default compiler.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows. In the following chapter, we provide a brief literature survey on software security, software security analysis, and automatic code

instrumentation techniques. In Chapter 3, we introduce *Team Edit Automata* as a new mathematical model for security property specification and present its implementation. Chapter 4 describes our solution to automatic code instrumentation - an extension to GCC compiler that can inject code at various security-sensitive program points. Chapter 5 provides the description of our integrated dynamic security analysis system and presents experimental results to elaborate our experience with it. We conclude the thesis in Chapter 6.

Chapter 2

Detection of Security Vulnerabilities in Source Code

This section summarizes the background of this thesis and the contributions made so far by other researchers in this field. It starts by outlining various security vulnerabilities in the source code of software, and then discusses program analysis and program instrumentation techniques.

2.1 Security Vulnerabilities in Source Code

A security vulnerability is defined as "a defect which enables an attacker to bypass security measures" [52]. The National Vulnerability Database of the National Institute of Standards and Technology [12] enlists eight classes of the causes to security vulnerabilities. These are summarized in [18] as follows:

- Input Validation Error (IVE), including boundary condition error and buffer overflow

error. This class of vulnerabilities result from program's failure to identify incorrect input or illegal memory access.

- Access Validation Error (AVE). AVE vulnerabilities is more commonly known as privilege escalation, where a user is given a higher access level than required.
- Exceptional Condition Error Handling (ECHE). These vulnerabilities include failure to respond to unexpected data or conditions.
- Environmental Error (EE). EE vulnerabilities surface when software runs in specific, usually harsh, execution environment.
- Configuration Error (CE). These vulnerabilities are the consequence of incorrect settings of the software or its execution environment.
- Race Condition Error (RC). RC vulnerabilities are specific to multi-processing and multi-threading software applications. They are the result of improper sequence or timing of multiple processes/threads.
- Design Error (DE). These vulnerabilities are the result of improper design of the architecture of the software.
- Others, i.e. vulnerabilities that do not fit in any of above categories. An example vulnerability of this type is improper choice of weak encryption algorithm.

2.2 Software Vulnerability Detection

Researchers and software engineers have developed many vulnerability-detection techniques in order to automatically evaluate and assure the security in software programs. Depending on whether the approach involves executing the programs, these techniques fall in two categories - static program analysis and dynamic program analysis. In this section, we present various vulnerability-detection techniques in these two categories. Our focus is mainly on: 1) How the tested security properties are specified in these techniques; 2) How violation of desired security properties is detected.

2.2.1 Static Program Analysis Techniques

Static analysis is the process of evaluating software or a software component based on the syntactic and semantic information inferred from its form, structure, content or documentation without program execution [16]. It has been applied to the detection of programming errors, violations of development standards, and other problems, which are not necessarily relevant to security. Examples of traditional application of static analysis include the identification of coding standard non-compliance, uncaught runtime exceptions, redundant code, unreachable code, etc. The growing interest in software security assurance in the past decades introduced new application of static analysis to vulnerability detection. For example, there are static analysis tools [32, 49, 84] that detect potential memory leaks, null-pointer dereference, illegal type-casting, vulnerability to SQL-injections, insecure file accesses, etc.

A non-exhaustive list of existing static security analysis tools (extended from the survey

in [83]) is shown in Table 1. These tools use various static program analysis techniques to detect security vulnerabilities in software. The following paragraphs present a brief overview of these techniques.

Pattern Matching Technique

Our discussion starts by presenting the pattern matching technique. For some programming languages, known vulnerable library functions exist that can result in function algorithm attacks. For example, the library functions such as `gets()`, `scanf()`, `sprintf()`, `strcat()`, `strcpy()`, and `vsprintf()` provided by the C programming language are known to be exploitable with specially crafted arguments [76]. The pattern matching technique works by searching source code for the statements that calls these functions and checking their validity through matching them to the pre-defined patterns.

PSCAN implements this technique in order to detect format string vulnerabilities in C programs [34]. It defines the legal and illegal patterns of calling functions of the `printf()` and `scanf()` family. It then searches the source code for statements calling these functions and matches them to the pre-defined patterns - those matching the illegal patterns are reported as warnings and errors.

The pattern matching technique defines security properties and their violation conditions in string patterns. However, only few security properties can be specified and checked with string patterns. Temporal properties, which involve program actions taken at different time during execution, cannot be correctly checked by matching patterns at a single program point. Consequently, tools using this technique only check a limited number of

Name	Focus	Target Language	Reference
BOON	buffer overflow	C	www.cs.berkeley.edu/~daw/boon
CodeWizard	Dangerous code constructs	C/C++	www.parasoft.com
ESC/Java2	Common runtime errors	Java	kind.ucd.ie/products/opensource/ESCJava2/
Flawfinder	Function calls, gettext libraries	C/C++	www.dwheeler.com/flawfinder
Illuma	Uninitialized variables, memory use and pointers	C/C++	www.reasoning.com
ITS4	Function calls, potential buffer overflows	C/C++	www.digital.com/its4
Klocwork	General-purpose static code analysis	C/C++/Java	www.klocwork.com
LDRA Testbed	General-purpose static code analysis	Ada/C/C++ Java/other	www.ldra.co.uk
MC Checker	Violation of temporal safety properties	C	www.stanford.edu/~engler/
xgcc & Metal	General-purpose static code analysis	C	www.stanford.edu/~engler/
MOPS	Violation of temporal safety properties	C	www.cs.berkeley.edu/~daw/mops
PC-lint	General-purpose static code analysis	C/C++	www.gimpel.com
PSCAN	Format strings	C	www.striker.ottawa.on.ca/~aland/pscan
RATS	Common security flaws	C/C++/Perl Python/PHP	www.securitysoftware.com
SLAM	General-purpose static code analysis	C/other	research.microsoft.com/slam/
Splint	buffer overflow, format errors	C	www.splint.org

Table 1: Static Security Analysis Tools

security vulnerabilities¹.

Induction from Program Annotation

This technique specifies security properties in terms of preconditions and postconditions. It requires the testee programs to be annotated with these properties before checking. For each checked function, the induction works by propagating the constraints defined in precondition, then verifying the accumulated constraints against the postconditions. Conflicts found during the induction process are considered to be security property violations.

Splint, previously known as LCLint, implements this technique. It requires security analysts to insert annotations as C program comments. These annotations are associated with global variables, structure fields, function parameters and function return values. Splint analyzes source code by resolving preconditions using postconditions of previous statements [14,41].

Meta-Compilation (MC) checker [20,93] also takes this approach. With user supplied annotations in the source code, it conducts a "flow-sensitive, bottom-up, inter-procedural analysis" [93] to propagate and induce the constraints defined in annotation. The analysis process starts with intra-procedural analysis. For each function, annotation defining its preconditions are induced and propagated on each path of the control flow graph (CFG) [17] to create a summary of the function. An inter-procedural analysis follows by propagating the summary from the callee to the caller at each function call site.

¹Efforts were made by RATS [79] to detect Time-Of-Check-Time-Of-Use (TOCTOU) race conditions using greedy pattern matching technique. However, the solution results in frequent occurrence of false positives according to [87].

The annotation induction technique has two advantages. Firstly, its induction simulates program execution statically. Thus, it is able to check temporal properties. Secondly, the checked security properties are extensible. This means that security analysts are able to define program-specific security properties. However, the technique suffers one disadvantage that inserting annotation to source code may require a significant effort for the million-line-of-code modern software. The required extensive human involvement makes it non-scalable for testing large-scale software.

Extended Type Systems in Compilers

Another approach to detect security vulnerabilities in software relies on extended type systems that are implemented in a compiler in order to detect vulnerable flaws in source code at compilation time. Briefly, type safety is a property attributed to some programming languages [70].

Newer programming languages like Java and C# are designed with type safety in mind. Programs written in such languages are immune to certain security vulnerabilities, such as illegal memory access due to improper type casting. However, many legacy software programs are written in non-strong-typing programming languages, such as C and C++ [70]. Some of these programming languages are still heavily used to develop new software programs. As a consequence, researchers have made efforts to introduce extensions to the existing type systems of these languages.

CCured [65] is a tool that implements this technique. It enforces a strong type system to "find a simple proof of memory safety for the program" [65]. In [86], another strong type system is proposed to statically detect type cast errors and temporal memory errors in

C source code.

Extending type systems for programming languages and enforcing it at compilation time is a useful technique to check certain security properties. However, it suffers three major drawbacks. Firstly, since the security properties (defined by the type system) are built in the compiler, it prevents type-checking user defined security properties. Secondly, the effort of implementing a type system in a compiler involves non-trivial effort. Lastly and most importantly, static type checking can only detect a limited set of security properties. As studied in [91], standard type safety properties only include *memory safety* (programs only access memory allocated to them), *control flow safety* (programs only execute valid code obeying execution sequence specified by their source code), and *abstraction preservation* (only abstract data types that are allowed by programs' interfaces are permitted). Although efforts were made to write type systems to enforce access control and other properties, the proposed technique [91] requires dynamic type checking and results in a non-standard type system (a system that encodes Security Automata [77]).

Static Program Execution Observed by Property-derived Automata

None of the static program analysis techniques introduced so far can be used to automatically check all temporal security properties. The pattern matching and extended type checking techniques are both limited to detect specific security properties. The annotation induction technique can be used to check more security properties. However, it can hardly be an automatic vulnerability detection solution due to its requirement of tedious program annotation.

In the `xgcc` analysis engine for Metal language [30], Engler et al. introduced a new

methodology of statically checking programs for security vulnerabilities. Their solution laid out "a general, extensible framework ... allowing the checking of a broad range of system-specific properties" [30].

The Metal/xgcc tool provides users with two utilities. Firstly, the Metal language is a C-like high level language designed to specify security properties. To be specific, the language defines key words and operators that allows users to define security properties in finite state automata. For example, the operator (`==>`) defines a transition from one state to another.

Secondly, xgcc is a fully automatic analysis engine that takes the user-defined security properties in Metal and the source code as input, and conducts full-path-coverage analysis of the source code. The engine parses the source code to produce a control flow graph of the testee program. It then makes a depth-first traversal of the control flow graph to visit each node of the graph. While traversing, it instantiates objects of finite state automata according to the user-defined security property specifications, forces the automata to make state transitions at tree nodes representing security-sensitive program actions, and reports errors when the automata reach error state.

In their publication [30], Engler and his colleagues demonstrated that the Metal/xgcc tool can automatically detect memory management and interrupt enabling/disabling flaws in Linux kernel. Therefore, their methodology of statically executing programs with security-sensitive actions observed by property-derived automata is a candidate solution to automatically check extensible security properties.

Model Checking Software

In the past few years, the application of model checking techniques [35] to program security analysis was studied by many researchers. Some well-known security analysis tools such as MOPS [32] and Klocwork [11] use this technique. The success of Klocwork (having won SD Times Award for three consecutive years [85] and InfoWorld 2007 Technology of the Year Award [56]) shows the strength of this technique. The following paragraphs introduce MOPS with more details, as Klocwork is proprietary software with little technical information exposed.

MOPS stands for Model Checking Program for Security Properties. It has been used to model check "an entire Linux distribution for security violations" [78] as well as one million lines of other C source code [31]. The security properties it checks are specified in Finite State Automata [50], which is very similar to Security Automata [77]. While secure program actions may transit the automata from one legal state to another, insecure program actions cause the automata to transit to an error state. Hence, MOPS focuses on detecting violations of temporal safety properties [32].

The tool works in two phases. In the first phase, it parses C source code and transforms the testee program into a pushdown automaton [50]. In the following phase, the model checker intersects the pushdown automaton with finite state automata, which defines the security property, to build a pushdown system [32]; it then traverses the pushdown system in search of reachable error state. When such a reachable error state is detected, it reports the security property violation with the offending path in source code.

Introduced as a validation technique, model checking is a competent technique in software security analysis. Since model checkers work with abstracted models of program states, which are usually written in checker-defined languages, the model checking engine is independent of the programming languages of the checked source code. An existing model checker only needs a parser to support a new programming language.

Summary of Static Program Analysis Techniques

Static program analysis for security evaluation discussed so far is appealing in practice due to the following facts.

Firstly, it has low runtime overhead. Static source code analysis does not require program execution. Many tools only parse and analyze the source code once to accomplish the security evaluation. Therefore, comparing to dynamic program analysis, which requires a program to be executed for as many times as the selected coverage criterion is satisfied, the runtime overhead of static source code analysis is much less.

Secondly, it is sound [39], since most static program analysis reasons over all possible runtime behaviors of the checked programs, their results describe the programs' behaviors regardless of their input or execution environment. Many static analysis techniques are based on existing formal methods, e.g. type system, model checking, and tree traversal algorithms. Accordingly, analysis engines that base their implementation on these methods guarantees to satisfy certain desired coverage. For example, the xgcc analysis engine supporting Metal language implements the depth first tree traversal algorithm to achieve full path coverage of the source code [30]. The model checker used by MOPS checks all the possible sub-states of the supplied model [32].

Lastly, it supports analysis of unrunnable programs. Dynamic program analysis necessitates executable programs or modules. On the contrary, static analysis does not execute the checked source code. Security analysts and software developers need not wait until a runnable program is available to start the security evaluation. Checking unrunnable programs is not only convenient but also necessary, because it is not always possible to compile source code into executables. For example, static or shared libraries used in the source code may not be available to the security analysts. In other cases, the specific hardware and/or operating system requirement of the program's execution environment may not be easy to set up.

The main limitation of static program analysis results from the approximation that is performed during analysis [39]. Static program analysis, such as those implemented in CCured, xgcc and MOPS, usually builds a model of the program state and reasons on how the program reacts to it. In order to maintain the soundness of the analysis, it must reason on all the possible executions of the programs. However, keeping track of all runtime states of a program is not always feasible. For example, an integer variable in C source code may have 2^{32} different runtime states; hence, the total number of runtime states of a program could be astronomical. Accordingly, static program analysis usually reasons on an abstracted model that drops some information. Taking integer variable for example again: static analysis such as symbolic execution [55] uses domain, denoted by the minimum and maximum values, to replace the concrete value states of an integer. Consequently, the analysis produces approximate and conservative results with possibly many false positives.

Name	Focus	Target Language	Reference
Dmalloc	Dynamic memory management flaws	C	dmalloc.com/
Eraser	Race conditions	Java	doi.acm.org/10.1145/265924.265927
EXE	General-purpose dynamic code analysis	C	www.stanford.edu/~engler/
fuzz + ptyjig	Pointer and array dereferencing, buffer overflow	C	doi.acm.org/10.1145/96267.96279
Insure++	Runtime memory errors	C/C++	www.parasoft.com/
JNuke	General-purpose dynamic code analysis	Java	www.schuppan.de/viktor/publications.html
Purify	Runtime memory errors	C/C++/Java C#/VB.NET	www-306.ibm.com/software/awdtools/purify/
Tester's Assistant	General-purpose dynamic code analysis	Java	portal.acm.org/citation.cfm?id=829503.830095
Valgrind	Runtime memory errors	C/C++/Java Perl/Python Fortran/Ada	valgrind.org/

Table 2: Dynamic Security Analysis Tools

2.2.2 Dynamic Program Analysis Techniques

No single fault-detection technique can address all fault-detection concerns [94]. Static program analysis techniques, being approximate and conservative, cannot guarantee the correctness of the program. A parallel stream of research, called dynamic program analysis, addresses the problems and limitations of static program analysis. This type of analysis involves executing the testee programs and detecting security properties violations at runtime.

Table 2 lists some existing dynamic security analysis tools found in our survey. The dynamic program analysis techniques used by these tools are briefly introduced in the following paragraphs.

Heuristics-based Random Testing

In [64], Miller et al. developed a simple method to detect security vulnerabilities (mainly out-of-boundary memory access) in UNIX utility programs. Instead of doing sophisticated static data flow analysis or runtime boundary checking, they developed a tool called Fuzz to generate variable-length streams of random characters. These randomly generated large string streams were given as input to the tested UNIX utilities. They found that "...the failure rate of utilities on the commercial versions of UNIX ... tested (from Sun, IBM, SGI, DEC, and NeXT) ranged from 15-43%" [63, 64]. The program flaws mainly resulted from illegal pointer manipulation and array dereference.

Using heuristics-based random test suite to crash the testee program is easy to implement and deploy. Miller's work demonstrated that applying this technique can sometimes find hidden flaws in the program. However, this technique does not guarantee soundness of the analysis results, i.e. the randomly generated test suites cannot assure that all the violations of security properties in the program be triggered. Moreover, this technique can only check a limited number of security properties. Take Miller's approach for example: Security vulnerabilities are detected only when the testee program halts or freezes upon a specific input string. No explicit definition of property violation conditions is provided or considered. Consequently, applying this technique alone can only detect those security vulnerabilities that immediately result in program halt or freezing.

Perturbation of Program Execution Environment

The heuristics-based random testing technique presented above uses specially crafted inputs to crash the program in order to detect vulnerabilities. In fact, program's execution environment can also be perturbed to crash it. For example, a dynamically linked program may behave normally when the shared library is available for dynamic linking. However, if the library is missing, the program may run with partial functionality or even crash. In a worse scenario, malicious users may replace the library with their hacked version, fooling the program to execute unintended code fragments.

In their book [92], Whittaker and Thompson discussed in details how to exploit software through execution environment perturbation. Particularly, they demonstrated with examples how to break security through attacking the software dependencies, adding constraints on memory and disk usages, forging data sources, etc. A utility software program called Holodeck was also developed to help security analysts crack the testee program through modifying its execution environment.

Testing software security with perturbed execution environment and input is especially useful when source code of the testee program is not available, because analysis can be done solely on executables. However, it is not a suitable solution to automatic security vulnerability detection, because execution environment perturbation is system specific. Additionally, such perturbation requires expert knowledge of the environments and their association to security.

Automatic Software Fault Injection for Security Testing

Software fault injection (SFI) is "an analysis technique that simulates faults and errors in order to see what impact they have" [89]. The aforementioned techniques of random testing and execution environment perturbation can both be considered to be specialized implementation of SFI. They are common in one aspect - both inject faults external to the program, either on its input or on its environment.

Ghosh et al. [44] came up with an automatic SFI implementation that injects faults in source code of the program. Their methodology was based on the premise that security vulnerabilities root in the flaws in source code. Their analysis focused on identifying "security-fatal" flaws in source code. To achieve this goal, they used automated fault injection technique to simulate flaws in various program points and used program monitors to detect runtime assertion failures, which reflect breach of security policies.

Particularly, they designed the Adaptive Vulnerability Analysis (AVA) algorithm [90] to dynamically execute the program, inject anomalous events during program execution, and determine if a security violation has occurred. The algorithm was implemented in Fault Injection Security Tool (FIST) [44]. The tool can automatically perturb values stored in the variables of primitive data types and simulate buffer overflows on program stack.

The automatic SFI-based security testing solution from Ghosh et al. has one major drawback. Since the security violations are triggered by perturbed software execution at source code level, there is no deterministic answer to whether these violations are really exploitable. For example, a pointer may be always checked against `null` before passed to the callee function. If the pointer is perturbed to a null pointer in the callee function, which

in turn crashes the program, we are not certain whether the crashing scenario ever occurs in reality.

Program Monitoring

The three techniques introduced so far mainly focus on triggering security violations when program executes. As a consequence, program halting and assertion failure only manifest runtime errors, but do not indicate which security properties are violated during execution. Other tools listed in Table 2 use program monitoring or runtime checking to detect security violations. These tools differ mainly in how and where the runtime monitoring is performed.

Dmalloc (the Debug Malloc Library) [2] is a drop-in replacement for the system's `malloc()`, `realloc()`, `calloc()`, `free()` and other memory management routines. These substitutional routines are implemented to keep track of the program's dynamic memory management actions. Such bookkeeping information is used to track memory leak and detect out-of-boundary memory access.

Purify [74] and Insure++ [10] instrumented monitoring codes on the program to detect runtime memory management errors. Purify maintains a state code for each byte of memory. The instrumented code "traps" each memory access the program makes to check the state for access inconsistency, e.g. reading an uninitialized memory block. Purify implements the state code as a two-bit flag stored in a bit table. The state transition is hard-coded to reflect the trapped memory accesses made by the program. Insure++ essentially operates in the same way as Purify does. However, it explicitly uses finite state automata to keep track of the security-sensitive program states.

JNuke is a specially crafted Java Virtual Machine (JVM) that "allows backtracking and full access to its state" [19]. It is used as a sandbox to execute the testee program and monitor its runtime behaviors. Typically, it transforms the byte code loaded from the testee program into a reduced instruction set and does instrumentation on the transformed code. As such, JNuke serves as an automatic program instrumentation and execution platform, allowing various security analysis algorithms to be developed on top of it. For example, Artho et al. implemented the Eraser [75] algorithms on JNuke to detect race conditions in testee programs.

The success of Purify and Insure++ proves that detection of interesting security vulnerabilities can be achieved by monitoring program execution with state transition systems. In addition, JNuke demonstrates that extensible security violation detection system can be implemented using program monitoring. The checkable security properties range from traditional memory management flaws to more sophisticated multi-threading problems as Time-Of-Check-Time-Of-Use vulnerabilities.

Summary of Dynamic Program Analysis Techniques

Dynamic program analysis has two advantages, which are complementary to static approaches. Firstly, it is accurate because security violations are triggered by concrete program execution. Secondly, it can check certain types of security vulnerabilities (e.g. memory management and pointer arithmetics) more efficiently.

Its major drawback is the lack of soundness, because it is difficult to generate test suites that execute all paths of a program. A preliminary solution has been separately proposed in [28] and [45].

That being said, dynamic program analysis is currently applied in the following ways.

1. It can be implemented in debugging tools. Dmalloc, Purify, Insure++ listed in Table 2 fall in this category. All three tools specialize in detecting program vulnerabilities related to memory management and pointer manipulation.
2. It can work synergically with static analysis tools. The complementary advantages of dynamic analysis over its static counterpart have been realized in some synergic security analysis tools. CCured is such an example. It inserts dynamic checking only at program points where static program analysis fails to reason correctly.
3. It can work as an extensible platform for security analysis. The only application we found during our survey is JNuke, although it is not a complete security analysis tool.

2.3 Program Monitoring and Instrumentation

2.3.1 Program Monitoring

Many dynamic program analysis tools need to observe program execution and gather information in order to analyze the testee programs' runtime behavior. This task can be carried out by monitoring tools.

Program monitoring can perform at different levels. At low level, hardware monitors extend the target system with specialized hardware architecture. For example, researchers included counters in their microprocessor's hardware to profile the execution of some pre-defined events [25, 71]. The advantage of hardware monitoring is its low overhead caused

by program monitoring and data recording. The major drawback of this approach is that monitors can only observe low-level data at restricted types of observation points [57].

Software monitoring is a more widely used approach to observe the program's runtime behavior. The monitoring functionality may run in parallel with the monitored program - for example, JNuke implements a sandbox to load, execute and monitor Java programs. It may also be incorporated into the monitored program, as in the case Insure++. In contrast to hardware monitors, software monitors can easily access high-level data, e.g. objects and structures in modern programming languages. Additionally, they can be designated to observe various program points such as predefined functions calls and access of particular variables. The drawback of this approach is its high runtime overhead.

2.3.2 Program Instrumentation

Implementing software monitoring techniques necessitates program instrumentation. The following paragraphs introduce various existing program instrumentation techniques.

Replacing Libraries

The simplest application to program instrumentation is realized without transforming the program. The technique involves only replacing the library used by the program. For example, the Dmalloc memory debugging library [2] replaces the standard C library to monitor memory management behavior of the monitored program. The monitoring functionality is included in Dmalloc library. Hence, the program only needs to be linked to the Dmalloc library to enable monitoring. This approach is clean and efficient. However, it prevents us from monitoring the program behaviors that are not associated with external libraries.

Therefore, it can only be used in very dedicated applications, as in the case of Dmalloc.

Preprocessor-assisted Source Code Transformation

C and C++ compilers call the preprocessor during the first phase of compilation to include external files into the compiled source code and perform textual substitutions that are defined by macros [76]. Simple code instrumentation can be implemented by utilizing the code transformation functionality of the preprocessor.

For example, Kranzlmuller [57] implemented preprocessor-assisted code instrumentation in his parallel program debugging tool. Figure 1 shows how the original `MPI_Isend` function is replaced by `mon_MPI_Send` in C programs. The definition of this substituting function is inserted into the program through the `include` macro.

```
#include <monitor.h>

#define MPI_Isend(buffer, count, datatype, \
                 dest, type, comm, request) \
                 \
                 mon_MPI_Send(__LINE__, __FILE__, MPI_ISEND, \
                               buffer, count, datatype, \
                               dest, type, comm, request)
```

Figure 1: Example of Macro-Assisted Code Instrumentation

Using macro-assisted source code transformation to perform code instrumentation is easy to implement. However, only few program points can be effectively instrumented with this approach. This is largely because that preprocessor cannot access the lexical structure and semantics of the instrumented program.

Parser-assisted Source Code Transformation

A parser is also a part of compiler. It builds the parse tree and abstract syntax tree of the program after source code is preprocessed. Therefore, it can access more information than the preprocessor can. The information includes the full syntactic structure and optionally the semantics of the program (depending on whether semantic analysis is performed).

Parser-assisted source code transformation can be very powerful. For example, the Puma library from AspectC++ project [13] can do sophisticated source code transformation at various program points. In fact, all the program weaving capabilities promised by the AspectC++ language are realized by the Puma library. An example implementation of parse-assisted code instrumentation is also available in the SUIF project from Stanford University [47].

This approach suffers one disadvantage that is universal to all program instrumentation approaches that rely on source-to-source code transformation. Generating instrumented source code requires extra file reading and writing, which increases compilation time. Such overhead is not tolerable by debugging tools and dynamic program analysis tools, which perform compilation fairly frequently.

Binary Wrapping of Object Code

The idea behind this approach is similar to macro-assisted code instrumentation. Instead of substituting program text, this approach replaces the bytes of the static object code of the instrumented program. Examples of binary instrumentation tools are ATOM [80] and JiTI [73].

The obvious advantage of this approach is that it requires no source code. However, its capability is restricted due to lack of the program's syntactic and semantic information. Moreover, it is dependent on the format of the static object code.

Instrumentation with AOP Weavers

Aspect Oriented Programming (AOP) is a new programming paradigm proposed for better modularized program implementation. In brief, AOP language allows developers to group-select a collection of points in the source code and to define what behaviors they intend to insert at these points. A software utility called *weaver* parses the language and transforms the program according to developers requirements. In [62], Mahrenholz et al. demonstrated that AspectC++ (an AOP extension for C++) can be used to do program instrumentation for debugging and monitoring.

As part of the research effort of this thesis, [24] studied the applicability of AOP as a code instrumentation tool for software security analysis. While AOP languages provide a very user-friendly means to specify code transformation requirements, they lack the syntax and weaver implementation to address low-level program points, e.g. variable access. At higher-level, program points concerning the data flow and control flow of the program are not fully implemented either.

Compiler-aided Code Instrumentation

Compiler-aided code instrumentation is actually an enhanced version of the parser-assisted approach. Instead of generating transformed source code from the modified parse tree or abstract syntax tree, it passes the tree to compiler's backend and immediately generates the

object code. The only overhead of such code instrumentation is the transformation of the program's intermediate representation. Hence, it is both a powerful and efficient way to perform code instrumentation. An example implementation is described in [29].

2.4 Summary

This chapter presents the state-of-the-art practices related to the research of this thesis.

We introduced different types of security vulnerabilities in source code. Their impact and persistence in software programs motivated researchers and security analysts to devise more effective countermeasures.

We provided a survey of static and dynamic program analysis techniques for security evaluation. We discussed the pros and cons of each approach. Moreover, we paid special attention to how security properties are specified in these techniques.

Finally, we presented several approaches to program monitoring and instrumentation, because the survey of existing dynamic analysis techniques indicates that program monitoring is an effective and scientific way to detect security vulnerabilities in programs. The study of existing program instrumentation techniques concludes that compiler-aided code instrumentation is a proper option to inject monitoring functionality in program.

Chapter 3

Security Property Specification

This chapter introduces a new mathematical model called *Team Edit Automata*, which captures the behavior of our program monitors for dynamic program analysis. The implementation of this model allows security analysts to specify security properties in a programmatic manner. Additionally, it constitutes the violation detection engine of the program monitors. We start with a brief introduction to the background and motivation of this new model in Section 3.1. We then present the definition and implementation in Section 3.2 and 3.3 respectively.

3.1 Introduction

3.1.1 Motivation

When concerning security evaluation of a software program, we must first ask three questions.

1. What are the security properties we intend to evaluate?
2. How can we formally specify these properties to guide our evaluation?
3. How can we detect flaws in the software program that violate the security property specification?

While the first question is somewhat program-specific, the last two can generally apply to security evaluation of any software programs.

We find analogous questions being asked and answered in the field of security enforcement, a sibling discipline of security evaluation. In that field, a ubiquitous technique is to monitor programs at execution and take remedial action when the programs violate a security policy. Researchers have introduced a variety of mathematical models [59, 77] of security enforcing program monitors and made significant efforts on defining the class of enforceable security policies by these monitors. Their models differ mainly in the monitors' remedial capabilities, including halting program execution, suppressing and inserting program actions. *Edit automata* [59], which combines all these capabilities, is proven to be able to enforce all security properties. Therefore, we consider *Edit Automata* to be a candidate model for formal specification of security properties.

However, the *Edit Automata* model is insufficient for the security evaluation purpose in two aspects. Firstly, the model is designed to enforce individual security properties. It does not explicitly define the enforcement action when different remedial actions are suggested by multiple program monitors, which concern the same program action. As for security evaluation, we require deterministic error condition to be defined based on all the security properties to be evaluated in a software program. This suggests the idea of considering and

synthesizing the output of these automata.

Secondly, the high interactiveness in modern software systems suggests that the monitoring automata be interactive as well. For example, dynamic memory and pointers are two tightly-paired components in C/C++ programs - deallocation of a memory block on the heap instantly invalidating all pointers referencing to it. Behind this example lies a star topology, where a dynamic memory block correlates to multiple referencing pointers. We want to generalize the interactiveness so that any two sets of monitors may interact, similar to the static program execution algorithm implemented in MeTaL/xgcc [30].

Both aspects of the aforementioned insufficiency can be addressed by combining the correlative individual *Edit Automata* into a single unit and allowing them to interact.

3.1.2 Related Work

Using finite state machines to describe security properties and using hierarchically structured automata to model the interaction among software components are both active research threads when considered in their own disciplines.

Monitoring software execution and taking remedial reactions to the violation of security properties is a ubiquitous technique used by researchers in the field of security enforcement. Schneider [77] introduced *Security Automata* to halt program execution when the programs' behavior violates security properties. He also characterized the class of security properties that can be enforced by *Security Automata* to be safety properties. Kim, Viswanathan and others [53, 88] explicitly added computability constraints on the safety

properties being enforced. Ligatti, Bauer, and Walker [22, 59] extends the enforcing capability of the program monitors by introducing *Edit Automata* as a sequence transformer. Recently, Talhi, Tawbi and Debbabi [27] introduced *Bounded Security Automata* and *Bounded Edit Automata* by adopting more precise abstractions and discussed their applicability to limited-memory systems. Along with the theoretical studies, various security monitoring systems [21, 22, 36, 38, 40, 51, 54, 88] have been implemented to allow arbitrary code execution upon possible violation of security properties.

In other disciplines such as the studies of distributed and concurrent systems, groupwares, and component-based systems, researchers use automata theory to construct formal models to specify the interaction within the systems. Lynch and Tuttle [60, 61] defined *Input/Output Automata* to model distributed and concurrent systems with different input, output and internal actions. Alfaro and Thomas [33] introduced *Interface Automata* for the specification and validation of systems communicating through their interfaces. Ellis and others [23, 37] introduced *Team Automata* as a mathematical model of the groupware systems. In particular, their model is defined as a hierarchical structure composed of multiple component automata responding to shared actions. Brim, Cerna and others [26] derived their *Component-Interaction Automata* from the *Team Automata* model in order to verify the behavior of component-based systems. Their model provides specification of how component automata are bound together and how the communication among components is carried out.

3.1.3 Security Property vs. Security Policy

To sum up our introduction, we differentiate security property and security policy in the following paragraphs. Their difference explains why the scope of this thesis is restricted to dynamically detecting violations of security properties only.

A security policy is a predicate P on sets of executions. A set of executions of a particular program satisfy a policy P if and only if every execution in the set satisfies the predicate P . Formally, “ Σ satisfies policy P ” if and only if $\forall \varepsilon \in \Sigma : P(\varepsilon)$ where Σ denotes a set of execution and ε denotes a single execution.

A security property is a predicate \hat{P} defined exclusively on individual executions of a program. An execution of a particular program satisfies a property \hat{P} if and only if every action in the execution satisfies the predicate \hat{P} . Formally, “ ε satisfies property \hat{P} ” if and only if $\forall \sigma \in \varepsilon : \hat{P}(\sigma)$ where ε denotes a single execution and σ denotes a sequence of actions.

It is clear that security properties only apply to individual execution while security policies are cross-execution concerns. As program monitors only see individual executions of the programs, they can only model security properties without further extension. Therefore, in this thesis, we exclusively focus on testing security properties.

3.2 Team Edit Automata

In this section, we give the formal definition of *Team Edit Automata*. We start by presenting the definition of *Edit Automata* and discuss its enforcement power when single security property is concerned. We then present the *Team Automata* model. The model and some of

its variants demonstrate possible formalization of component-interactive systems. Finally, we present the definition of *Team Edit Automata* and explain the semantics of this model.

3.2.1 Edit Automata

Prior to the introduction of *Edit Automata*, Schneider [77] presented *Security Automata*. He uses *Security Automata* as sequence recognizers, such that they only emit sequences of actions that are in conformity with the enforced security properties. Upon receiving sequences of actions that violate the enforced security properties, the automata halt the program execution. According to his proof, *Security Automata* can enforce all safety properties (“nothing bad ever happens” [58]) and a limited set of security properties.

Ligatti, Bauer, and Walker [59] defined *Edit Automata* such that they can enforce all security properties. They view the program monitors as a sequence rewriter. Hence, rather than simply halting the program execution, their program monitors can suppress and insert actions to the programs. They introduced *Edit Automata* to model their program monitors and reasoned that their model can enforce all security properties.

An *edit automaton* E is a triple (Q, q_0, δ) defined with respect to some system with action set \mathcal{A} , where:

- Q is a finite or countably infinite set of states
- q_0 is the initial state
- $\delta : Q \times \mathcal{A} \rightarrow Q \times (\mathcal{A} \cup \{\cdot\})$ is a transition function

The semantics of δ is defined to be:

$$(q, \sigma) \xrightarrow{\tau}_E (q', \sigma')$$

- if $\sigma = a; \sigma'$ and $\delta(q, a) = (q', a')$ then $(q, \sigma) \xrightarrow{a'}_E (q', \sigma)$ (E-Insertion)
- if $\sigma = a; \sigma'$ and $\delta(q, a) = (q', \cdot)$ then $(q, \sigma) \xrightarrow{\cdot}_E (q', \sigma')$ (E-Suppression)

In above definition, a and a' denote individual actions while the symbol "." denotes empty sequence of program actions. σ and σ' denote sequences of actions. The symbol ";" denotes concatenation of actions, e.g. $a; \sigma'$ means an action a followed by a sequences of actions (represented by σ'). The action above the arrowed line ($\xrightarrow{\cdot}$) represents the one emitted by the automata.

An *edit automaton* can temporarily suppress a sequence of actions that may violate the security property it monitors. Hence, instead of allowing potentially illegal actions to take effect, the *edit automaton* records the sequence internally and waits for the action that can guarantee the sequence to be legal. If such action arrives, the automaton will emit all previously suppressed actions and continue to process the upcoming actions. This way, it reserves the semantics of the program actions and enforces the security property as well. If the action is absent, the automaton halts the program execution (by suppressing all future actions).

While *Edit Automata* are defined to model the program monitors for security enforcement, we consider them to be a mathematical model for describing security properties. An *edit automaton* defines all action sequences that satisfy the security property monitored by its corresponding program monitor. It can also recognize all the actions sequences that violate the security property. Hence, it divides all the action sequences of a software program into two partition classes - one includes all legal sequences and the other all illegal ones.

We can take advantage of this feature of *Edit Automata* to specify security properties for testing purposes.

In [22], Ligatti, Bauer, and Walker noticed the problem when multiple security properties are concerned of identical actions. Intuitively, there must be a way to tell which enforcement reaction defined by which program monitor should take effect in such scenarios. They addressed the problem by including “Policy Combinators” in their empirical implementation called *Polymer*. We consider their approach to be very suggestive. A general mathematical model for security property specification should be able to describe such scenarios. Hence, we need an architecture that can group multiple automata together.

3.2.2 Team Automata

Team automata [37] was introduced by Ellis as a mathematical model of groupware systems. The model defines a way where multiple collaborative component automata can be interconnected to form a team and where multiple teams can be interconnected to form a larger-scaled architecture.

A *component automata* C is defined as a 4-tuple $\langle Q, Q_0, A(C), \delta \rangle$, where:

- Q is a nonempty set of states;
- Q_0 is a nonempty set of initial states such that $Q_0 \subseteq Q$;
- $A(C)$ is an ordered triple consisting of three pairwise disjoint sets of actions $\langle in(A), out(A), int(A) \rangle$, where $in(A)$ defines input actions, $out(A)$ defines output actions, and $int(A)$ defines internal actions that are not externally observable;

- $\delta : Q \times A(C) \rightarrow Q$ is a state transition function.

The *component automata* are similar to ordinary automata except that their actions are classified into three categories - input actions, output actions, and internal actions. The categorization of actions can be viewed as an extra attribute of the actions. The team relies on this attribute to connect multiple automata. For example, two automata are connected if the output action from one component automaton is the input action to the other.

Given a set of component automata $\{C_i\}$, a team automata T is a four-tuple $\langle Q^T, Q_o^T, A(T), \delta^T \rangle$, where:

- $Q^T = \Pi(Q_i)$, where Π denotes the Cartesian products, is a nonempty set of states;
- $Q_o^T = \Pi(Q_{oi})$ is a nonempty set of initial states;
- $A(T)$ is an action signature that defines on each input action which component automata should simultaneously execute it and which should keep dormant;
- $\delta^T : Q^T \times A(T) \rightarrow Q^T$ is a transition function defined on all possible input actions to the team.

The team automata are defined, through their action signature $A(T)$, to be able to find all the component automata, which can execute an action from their current state and require them to execute it simultaneously. Accordingly, the team allows a single action to be broadcasted to multiple components. This behavior is very similar to what we intend to include in our model.

3.2.3 Team Edit Automata

The *Team Edit Automata* model combines the powerful enforcing capabilities of *Edit Automata* into the component-interactive architectural model defined by *Team Automata*. The resulting model is a team composed of one or multiple component edit automata. A *team edit automaton* connects its component automata through action signatures - definitions that designate the source and destination of actions.

Notation

The notation used in the following definitions is similar to that in [59].

We consider a software program as a set of program executions \mathcal{A} . We use the notations \mathcal{A}^* and Σ to respectively denote the set of all sequences of actions of a program execution and an arbitrary set of executions such that $\Sigma \subseteq \mathcal{A}^*$.

We use ε to represent a single execution and the symbol “.” to denote the empty sequence of program actions. We use σ and τ to define single sequences of actions and we use a and a' to denote single program action. We use the notation $\sigma; \sigma'$ to define the concatenation of two sequences of actions σ and σ' .

The arrowed line (\longrightarrow) denotes a state transition. If any symbols appear above it, they represent the actions emitted by the automata.

Component edit automata

We use the component edit automata to model program monitors for individual security properties. Each automaton specifies one security property of the program.

A *component edit automata* C is a 5-tuple $\langle Q, I, A, G, \delta \rangle$, where:

- Q is a nonempty finite or countably infinite set of automaton states;
- I is a nonempty initial state set such that $I \subseteq Q$;
- A is a set of actions;
- G is a set of guard conditions;
- δ is a partial function $A \times Q \times G \rightarrow Q \times A \cup \{\cdot\}$ for transition relation.

The symbol δ specifies the transition function for the automaton. Based on the automaton's current state, input action and guide condition, the function indicates whether the automaton should suppress the input action, insert action(s) in the output, or report errors. Transitions not defined by δ are considered to be errors. In such cases, the component automaton reports security violations to the team and remains in its current state.

It is worth mentioning that our *component edit automata* augment *Edit Automata* with a guard condition on the transition function. Without guard conditions, the state transition of *Edit Automata* is based only on the temporal properties of the input actions, constraining the expressiveness this model. On the contrary, our *component edit automata* can check context information before making state transitions. Experiment 1 described in Chapter 5 demonstrates the usefulness of guard condition in actual application.

The operational semantics of *component edit automata* is specified as follows:

$$(q, \sigma, g) \xrightarrow{\tau} (q', \sigma')$$

- if $\sigma = a; \sigma'$ and $\delta(q, a, g) = (q', a')$ then $(q, \sigma) \xrightarrow{a'} (q', \sigma)$ (insert)
- if $\sigma = a; \sigma'$ and $\delta(q, a, g) = (q', \cdot)$ then $(q, \sigma) \xrightarrow{\cdot} (q', \sigma')$ (suppress)
- if $\sigma = \sigma'; a; \sigma''$ and $\delta(q, a, g) = 0$ then $(q, \sigma) \xrightarrow{r; \sigma'} (q, \sigma'')$ (report flaw)

Accordingly, a component edit automata can insert more action(s), suppress the input action, and report possible flaws in the program.

The semantics of the “insert” and “suppress” operations is similar to the one defined in *Edit Automata*. An example application of the “suppress” operation for security testing is to suppress the second `free` action when double-freeing is detected. Since we are certain that such suppression prevents the testee program from crashing without perturbing its runtime states, the suppression allows us to detect subsequent vulnerabilities on the same execution path, instead of halting it.

The “report flaw” operation does not confirm but reports a possible flaw. 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, followed by all previous input actions. The output action “r” signals the team of a potential flaw in the program. The appended history of previous actions provides the context of the detected property violation.

The final judgment of the detection of flaws is delegated to the team, which can collect outputs from all correlated component automata (those respond to the same input action) and make judgments based on its own specification.

Team Edit Automata

A *Team Edit Automata* is composed of one or more component edit automata. It groups correlative security properties (as component automata) in a team, coordinates the interaction among these components and ensures that the team responds to program inputs with explicitly defined outputs.

In our model, we use action signatures to describe the shared actions among component

edit automata.

An action signature a^T is a 3-tuple $\langle \{C_j\} \cup \{-\}, a, \{C_k\} \rangle$ for $j, k \leq i$ where $\{C_j\}$ and $\{C_k\}$ are sets of component edit automata and a represents an action.

An action signature has the following two types:

- **Pipe:** $\langle \{C_j\}, a, \{C_k\} \rangle$ defines an output action a from any component edit automata in set $\{C_j\}$ to be passed as input to all component edit automata in set $\{C_k\}$;
- **Input:** $\langle \{-\}, a, \{C_k\} \rangle$ defines an action a sent to the team automata as input and recognized by all the component edit automata in set $\{C_k\}$.

Note that the action a is synchronously processed by all component automata in the set. This is why we call it a shared action. Semantically, the **Pipe** and **Input** action signatures combine individual component edit automata into a group. In our model, two individual component automata are in the same team either when they share a set of non-empty identical input actions (defined by an **Input** action signature), or when the output action of one component automaton is the input action of the other (defined by a **Pipe** action signature).

Input action signature correlates security properties (modeled by their corresponding component edit automata) that share concerns of a common set of program actions. For example, a program may have one file access policy and one user authentication policy implemented. The file access policy forbids any user except the system administrator from sending packets from a socket connection once s/he opens some sensitive local files. On the other hand, the user authentication policy allows any successfully authenticated users to send any packets of information to the network. In this case, a user trying to send some packets of information to the network is concerned by both policies. Our model correlates

these two policies into one team and provides additional specification with regard to how to respond to outputs from these two component edit automata. Hence, the specification of the expected implementation of the security properties in a software program can be accurately defined.

Pipe action signature correlates security automata whose concerned program actions may affect each other's state(s). In the same program as in above example, users may successfully authenticate themselves as the system administrator, open the sensitive local file, then start sending it over the network. To evaluate whether the program correctly implements the file access policy to deal with this scenario, we can either let the corresponding component edit automata to query the users' authentication information then judge the correctness of the implementation. This is feasible but ad-hoc in nature. Or, we can make a production of the component edit automata of the two policies and use the new production automata to monitor the implementation of both security properties. This is feasible as well but does not scale well. In our model, the interaction among the component edit automata is defined by allowing the output of some automata to be piped as input to others. Hence, no automata production is required and the model can generally apply to any software programs.

We now give the definition of *Team Edit Automata* as follows.

A *Team Edit Automaton* T is a 3-tuple $\langle \{C_i\}, A^T, \delta^T \rangle$, where:

- $\{C_i\}$ ($i \in \mathbb{N}$) is a nonempty set of component edit automata. The Cartesian product of their states and initial states constitutes the states and initial states of the team respectively;

- A^T is a set of action signatures $\{a_j^T\}$ ($j \in \mathbb{N}$);
- δ^T is a partial function $\{A \times \{A_k\}\} \rightarrow \{A_l\}$ ($k, l \leq i$) for flaw judgment and output transformation.

The function δ^T determines the team's observable outputs. It works in two ways:

Firstly, it defines what output the team should emit if multiple component edit automata emit different outputs upon an identical input action. In this case, δ^T is particularly defined as a partial function $\{A \times \{A_k\}\} \rightarrow \{A_l\}$ ($k, l \leq i$ and $r \notin \{A_l\}$). By explicitly defining δ^T for such conditions, testing engineers are able to specify the expected security property implementation more accurately. This is also a formalization of the “Policy Combinators” implemented in Polymer [22], which we discussed in section 3.2.1.

Secondly, we mentioned that component edit automata only report flaws they detect and the judgment of whether the flaws should be made observable is delegated to the team. δ^T is used to describe how the team should make such judgment. In this case, δ^T is particularly defined as a partial function $\{A \times \{r\} \cup \{A_k\}\} \rightarrow \{A_l\}$ ($k, l \leq i$) where r is the report-flaw output action from component automata. Therefore, if an action r is output by multiple component automata upon an input action a in A , δ^T defines the team's behavior of whether it should report the flaw(s) or not. Note that given the input action a , we are able to find all correlative component automata by looking in the definition of action signatures.

3.3 Implementing Team Edit Automata

3.3.1 Design Overview

Our implementation of Team Edit Automata is composed of a hierarchy of C++ classes.

Particularly, four classes are defined as following:

- Class `Event` provides an abstraction of program actions. It is supposed to be inherited by concrete subclasses in real application. Both team automata and component automata are triggered by `Event` for state transition.
- Abstract class `ComponentAutomata` corresponds to the *Component Edit Automata* in our model. It must be inherited by concrete subclasses in real application to represent various individual security properties.
- Class `TeamAutomata` implements various team management and collaboration functionalities. It allows components of the same team to interact with each other and guarantees to generate a team-wise response upon any program action.
- Abstract class `SuggestionSolver` implements the strategy design pattern [15]. In real applications, security analysts can implement system-specific suggestion solver to solve conflicts among security properties. We included a simple default suggestion solver in our implementation that always respects suggestions in a particular priority.

Figure 2 shows the collaboration graph of three main classes in our implementation.

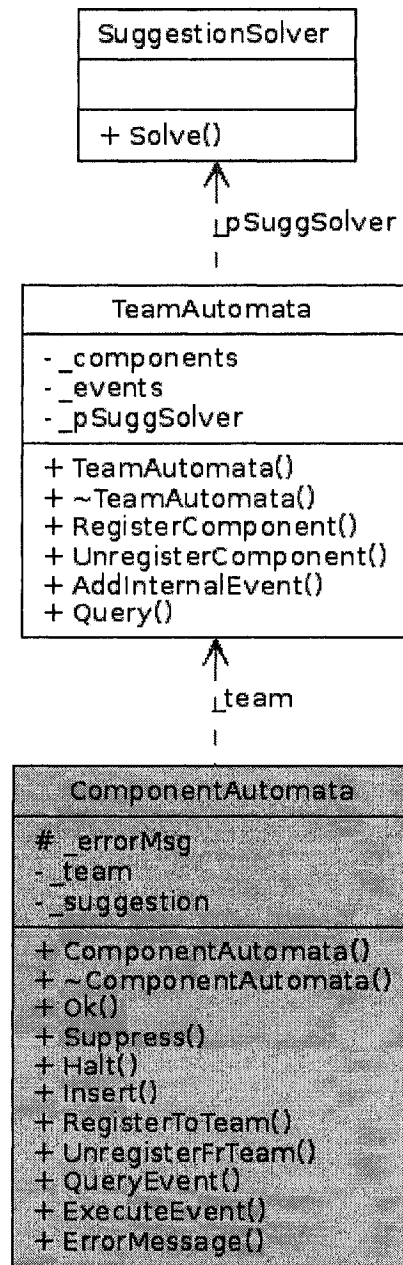


Figure 2: Collaboration Graph of Team Edit Automata Implementation Classes

3.3.2 Implementing Component Automata

We implemented component edit automata as an abstract class `ComponentAutomata`. Figure 2 shows its public interfaces.

Each `ComponentAutomata` object is associated with a team through the private member of `_team`. It can switch team through the `RegisterToTeam()` and `UnregisterToTeam()` interfaces.

We separated the component's state transition into two phases, namely `QueryEvent()` and `ExecuteEvent()`. In the first phase, i.e. `QueryEvent()`, a `ComponentAutomata` object responds to the input action with a suggestion and does not perform state transition. In the second phase, it makes the transition responding to the input action. The idea behind this design decision will be clear when we introduce the workflow of the `TeamAutomata` class.

We intentionally separated the state transition functionality from `ComponentAutomata`, so that any implementation of a finite state machine can be incorporated into our class hierarchy. In our integrated system (see Chapter 5.1), we used State Machine Compiler [72] to automatically generate C++ implementation of finite state machines and composed them in `ComponentAutomata` subclasses.

3.3.3 Implementing Team Automata

Class `TeamAutomata` implements the *Team Edit Automata*. As shown in Figure 2, a `TeamAutomata` object manages multiple `ComponentAutomata` objects through the `RegisterComponent()` and `UnregisterComponent()` interfaces.

The class also manages an event queue (i.e. `_events`) to store all the unprocessed events. Internal events sent between components can be enqueued through the `AddInternalEvent()` interface.

The task of solving conflicting suggestions is delegated to a `SuggestionSolver` object. This way, security analysts are free to define any conflict solving algorithm to be used by the team.

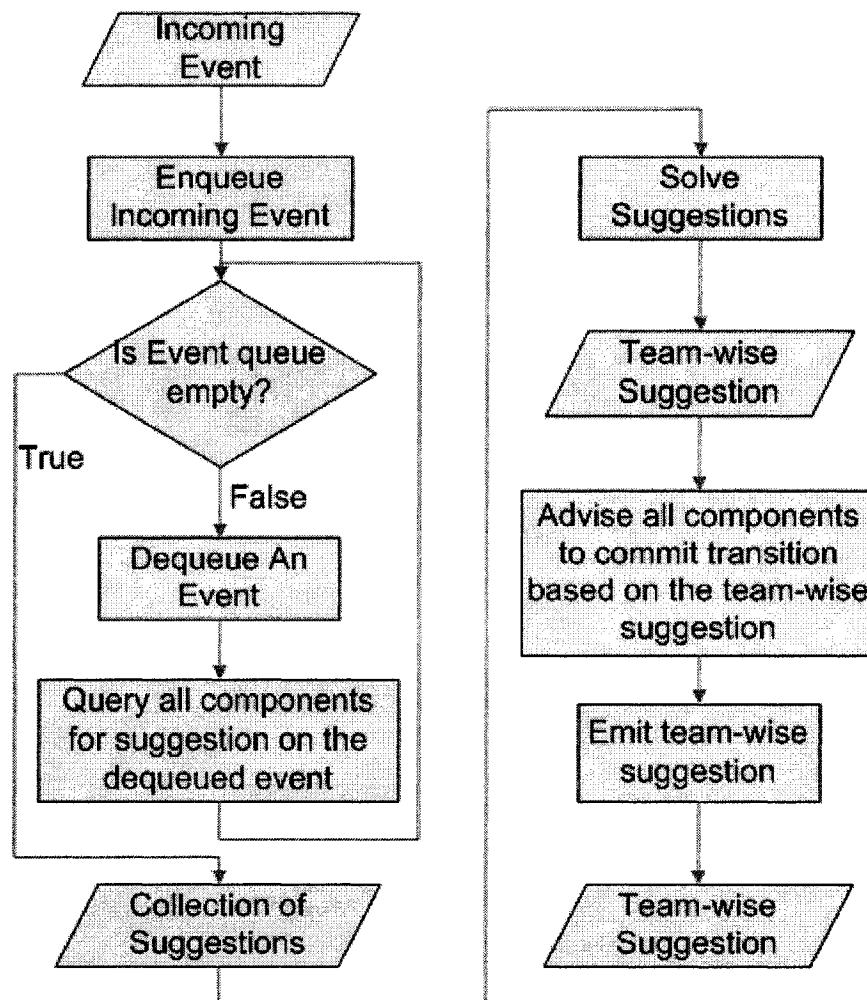


Figure 3: Workflow of `TeamAutomata::Query()`

Finally and most importantly, `Query()` is a team's public interface to the monitored program. During execution, the instrumented monitoring code frequently calls this function

for team-wise suggestions. Figure 3 shows the workflow of this function. Upon receiving an input action, the function broadcasts it to all the component automata and collect their suggestions. Some component automata may send internal actions to other components. Thus, the function keeps collecting suggestions until all the internal actions are broadcasted. It then calls the `SuggestionSolver` object to produce a team-wise suggestion based on all the collected suggestions from the components. This suggestion is emitted as output action after being broadcasted to all component automata to commit state transition.

3.4 Summary

In this chapter, we introduced *Team Edit Automata* as a mathematical model to capture the behavior of program monitors that check the security properties.

The research effort was motivated by the requirement of allowing user-defined security properties to be dynamically checked in a universal and scientific way. *Team Edit Automata* was introduced to specify all temporal security properties. It also provides additional definitions to: 1) model interactions between software components; 2) improve the expressiveness of the model by adding guard conditions to state transitions; 3) allow multiple security properties to be specified for a program without conflicts.

We presented the implementation of *Team Edit Automata* as a hierarchy of C++ classes. These classes provide a framework for programmatically writing security properties. In real applications, we provided graphic interface to facilitate property specification so that security analysts need not to code the properties from scratch rather draw property specifications with a visual aid. The introduction of this graphic interface (called *Security Property Editor*

in our integrated system) will be presented in Section 5.1.

Chapter 4

Code Instrumentation

The following chapter depicts the implementation of a GCC extension for code instrumentation. The text starts by explaining why we choose this approach. Section 4.2 introduces the functionality and user interface of the extension and Section 4.4 details its implementation.

4.1 Extending GCC for Code Instrumentation

We chose the approach of compiler-aided code instrumentation for the following reasons:

- Compiler knows the lexical structure and the semantics of the program being instrumented. It builds and analyzes the Abstract Syntax Tree (AST) and the control flow graph of a program. Therefore, we can precisely select the program point for code instrumentation.
- The existing compiler optimization can apply to the instrumented code, resulting in

efficient executable file. Optimization of the instrumented program is crucial to dynamic analysis. For a sounder analysis result, the instrumented program is usually executed as many times as needed to explore different execution paths of the program. Therefore, a better optimized program can reduce the analysis time.

- The program monitoring functionality is compiled as part of the instrumented program, also resulting in faster executables.

The reasons to why we chose to extend GCC instead of other compilers are listed below:

- GCC is the default compiler for many FOSS projects. As our goal is to evaluate security in FOSS projects, extending GCC facilitates easier integration to the build system of these projects. Code instrumentation can be performed on FOSS projects without modifying their Makefiles.
- GCC is a well developed and tested multi-platform compiler. It can cross compile software program for various hardware chips and operating systems.
- GCC is a collection of compilers for many programming languages, including C, C++, Objective C, Fortran, Java, Ada, and so on. The compiler uses a universal intermediate language called GIMPLE to represent programs written in all these languages. Therefore, we chose to perform code instrumentation at the GIMPLE level so that future extensions can be added to support code instrumentation for all these languages.

4.2 Introduction to the Extension

We have implemented an extension to the GCC compiler for C programming language. Our implementation is based on the GCC core distribution version 4.2.0 [5]. In addition to all the functionalities of the original GCC compiler, the extension can do automatic code instrumentation. The instrumentation process, i.e. where and what to instrument, is guided by an input file written in a specific language. So far, the extension can properly work on both UNIX and Linux systems.

4.2.1 Workflow Overview

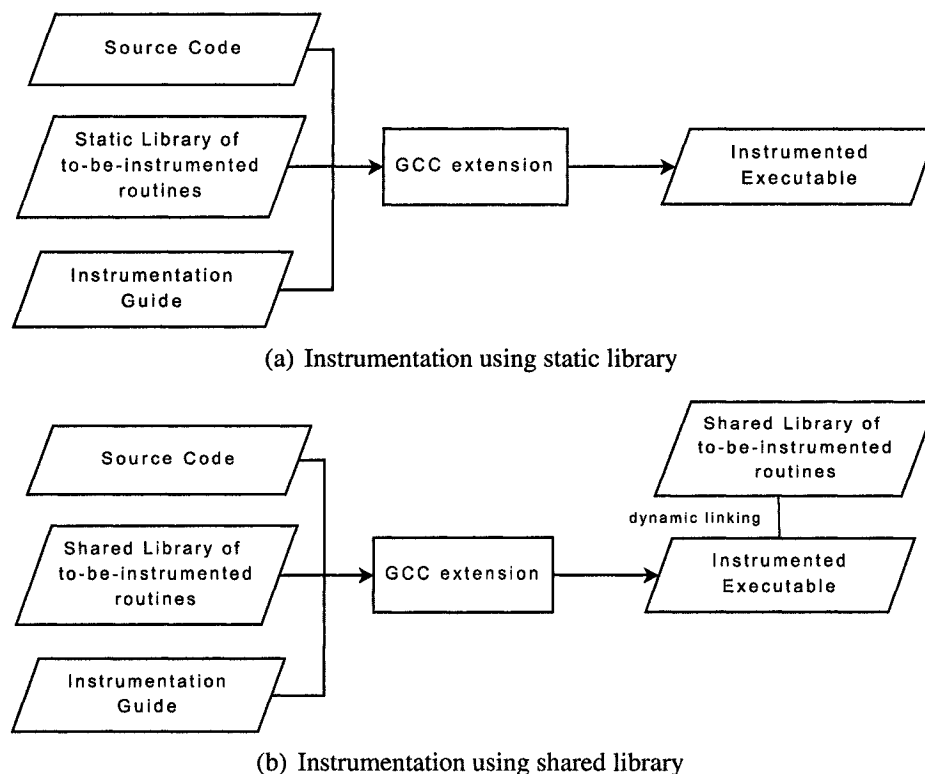


Figure 4: Workflow of the GCC Extension for Code Instrumentation

Figure 4 depicts an overview of the workflow of the extended GCC compiler. Generally, the compiler extension takes three inputs: the source code, an input file to guide the instrumentation process (mentioned as instrumentation guide in the following text when there is no ambiguity), and a library containing the subroutines to be instrumented in the program (mentioned as instrumentation library when there is no ambiguity). If only compilation is required, the extension outputs an .o object file. Otherwise, it outputs an executable file. In both cases, the output is instrumented with a behavior that is implemented in the instrumentation library.

Figure 4 also shows that our extension can work with both static and shared instrumentation libraries. This provides security analysts with the flexibility of choosing different linking options. Moreover, linking with shared libraries improves the usability of our extension. As we mentioned in Chapter 3, our program monitors are implemented in C++. Consequently, we need to wrap the program monitoring routines with C interfaces and dynamically link them in the instrumented program. The dynamic linking capability of our extension immediately solves this problem.

4.2.2 Functionality

The functionalities we chose to implement in our GCC extension are mainly oriented by the requirement of injecting monitors in the programs for the purpose of dynamic vulnerability detection. On the one hand, we carefully selected a set of program points where code can be instrumented. The chosen points are all considered to be security sensitive in C programs. On the other hand, the instrumented code can change the program's original control flow

at certain program point.

Program Points for Instrumentation

Our extension can instrument code at any of the following points in a C program:

- **Function call:**

C programs perform various tasks through calling functions (also known as sub-routines). Therefore, instrumenting code at function calls is crucial to check many security properties in programs. For example, the double freeing vulnerability in C programs can be detected by monitoring calls of `malloc` family functions and the `free` function.

In order to supply dynamic analysis with more information, we allow security analysts to optionally expose the arguments that are passed to a function call. Additionally, if the instrumentation is done after the call, the analysts can expose its returned value as well.

- **Function exit:**

If any program monitor is instantiated to track the intraprocedural context of a function, codes can be instrumented before it exits. Usually, the instrumented code summarizes the intraprocedural analysis result and releases the monitor at this type of program points.

- **Variable declaration, read and write:**

These program points are included in our selection because many security vulnerabilities are the result of invalid use of variables. In particular, some low-level security

vulnerabilities such as array out-of-boundary access have to be detected at variable level [24].

Variables are identified by their names and locations in the source code (i.e. source file name and line number). Our current implementation is able to identify all types of variables in C programs, including global variables, static variables, and automatic variables (also known as local variables).

- End of a variable's binding scope:

At this special type of program points, instrumentation code can release the program monitors instantiated to trace the states of variables. As for automatic variables, it corresponds to when the program execution is about to leave the scope where the variable is declared. In the case of global variables and static variables, it corresponds to when the program is about to exit.

- Pointer dereference:

We paid special attention to the use of pointers in C programs, because misuse of pointers can result in many security vulnerabilities. With this type of program points included, security analysts can monitor any specific pointer dereference action in the program.

Suppression, Insertion, and Halt

As mentioned in Chapter 3, we proposed a model of program monitors, which can suppress and insert actions in the monitored programs as well as halt them. Accordingly, we implemented our GCC extension such that the instrumented code can take all these proposed

actions.

- **Suppress actions:**

If monitoring code is instrumented before a security-sensitive statement, the program monitor may suppress the statement, depending on the security property it checks. More details on the implementation of this feature is presented in Section 4.4.

- **Insert actions:**

Inserted program actions are included in the shared or static library that contains program monitoring and analysis routines. Security analysts are free to add any desired behavior in the library and have them linked into the instrumented program.

- **Halt program:**

Halting the instrumented program is implemented through instrumenting a call to `exit` in the program.

4.2.3 User Interface

Invocation of the Instrumentation Extension

There are two ways by which security analysts can enable our GCC extension to do code instrumentation:

- **Using command line options:**

Our extension added a particular command line option to the GCC compiler. To enable the extension to instrument code, security analysts include the following option when invoking our extended version of GCC at command line:

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

"**-ftree-security-instrument**" advises our GCC extension to enable the instrumentation functionality. "**instrumentation_guide_input_file**" is the input file of instrumentation guide.

In addition to the above option, security analysts need to add the linking options to specify the library that contains program monitoring routines. This is done through GCC's "**-l**" and "**-L**" options.

Below is an example of a complete command that performs the invocation:

```
gcc -L\absolute\dir\to\SecInstrLib -lSecInstrLib  
-ftree-security-instrument=instrumentation_guide_input_file  
source_file.c
```

- Defining environment variables:

Using command line options to enable code instrumentation is probably a convenient means to compile a separate source file or write a Makefile [9] of a software project from scratch. As for FOSS projects, whose Makefiles are either automatically generated by autoconf utility [1] or distributed with source code, it still requires some human effort to modify the Makefiles before code instrumentation can be done. The effort of such modification is determined by the complexity of the Makefiles.

Since our ultimate goal is to seamlessly integrate code instrumentation into FOSS projects' build system, we implemented the following means - through defining two specific environment variables.

FTREE_SECURITY_INSTRUMENT is the first environment variable that needs to

be defined. Its value is expected to be the absolute directory and file name of the input file of instrumentation guide. On Linux systems, this can be done through the following Bash command [7]:

```
export FTREE_SECURITY_INSTRUMENT=  
    \absolute\dir\to\instrumentation_guide_input_file
```

SHARED_LIB_NAME is the other environment variable that must be defined. It should contain an absolute directory and file name of the library containing program monitoring routines. An example of its definition of Linux Bash command line is as following:

```
export SHARED_LIB_NAME=\absolute\dir\to\SecInstrLib
```

Consequently, building a Makefile based FOSS project with code instrumentation can be achieved with three commands on Linux systems as following:

1. **export FTREE_SECURITY_INSTRUMENT=**
 \absolute\dir\to\instrumentation_guide_input_file
2. **export SHARED_LIB_NAME=\absolute\dir\to\SecInstrLib**
3. **make**

Low-level Instrumentation Guide

The instrumentation performed by our GCC extension is guided by the instructions in an input file. These instructions are called a low-level instrumentation guide, because they are defined mainly to facilitate fast and efficient processing in the compiler.

Each instruction is a line of string composed of eight fields delimited by spaces. The

order, meaning and format of these fields are listed in Table 3.

Order	Purpose	Format and Value
1	Scope of the concerned program point	"function_name" for a function scope or "*" 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 "*"; where lineNum 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 3: Fields in the Instrumentation Guide Input File

Accordingly, an example instruction in the following line

```
main 1 16 test.c::32::i SecInstr_VarDecl 1 0 0
```

advises our extension to instrument a call to function `SecInstr_VarDecl()` after the declaration of variable `i` at line 32 of the source file "test.c" in the `main()` function scope.

The return type of `SecInstr_VarDecl()` is `int` and the last two fields are not used in

this example.

High-level Instrumentation Guide

The low-level instrumentation guide presented above is obviously not easy to write or read. We therefore defined a simple and friendly language for security analysts to write the instrumentation guide. The grammar of this language is presented in Appendix A.

Accordingly, to achieve the same instrumentation goal as in previous example, the analysts can write:

```
after declvar (test.c::32::i) inject "SecInstr_VarDecl";
```

This is certainly more readable and easier to write.

In actual application, the analysts need only write the high-level instrument guide. They can then compile it into low-level guide before invoking our GCC extension.

4.3 GCC Internals

This section introduces the GCC internals [42] in brief. Two topics are covered to help readers better understand our implementation. Section 4.3.1 introduces the architecture and compilation phases of GCC. Section 4.3.2 presents the GENERIC and GIMPLE intermediate languages.

4.3.1 GCC Architecture and Compilation Phases

Figure 5 shows the architecture of GCC and the phases of compilation in GCC version 4.2.0.

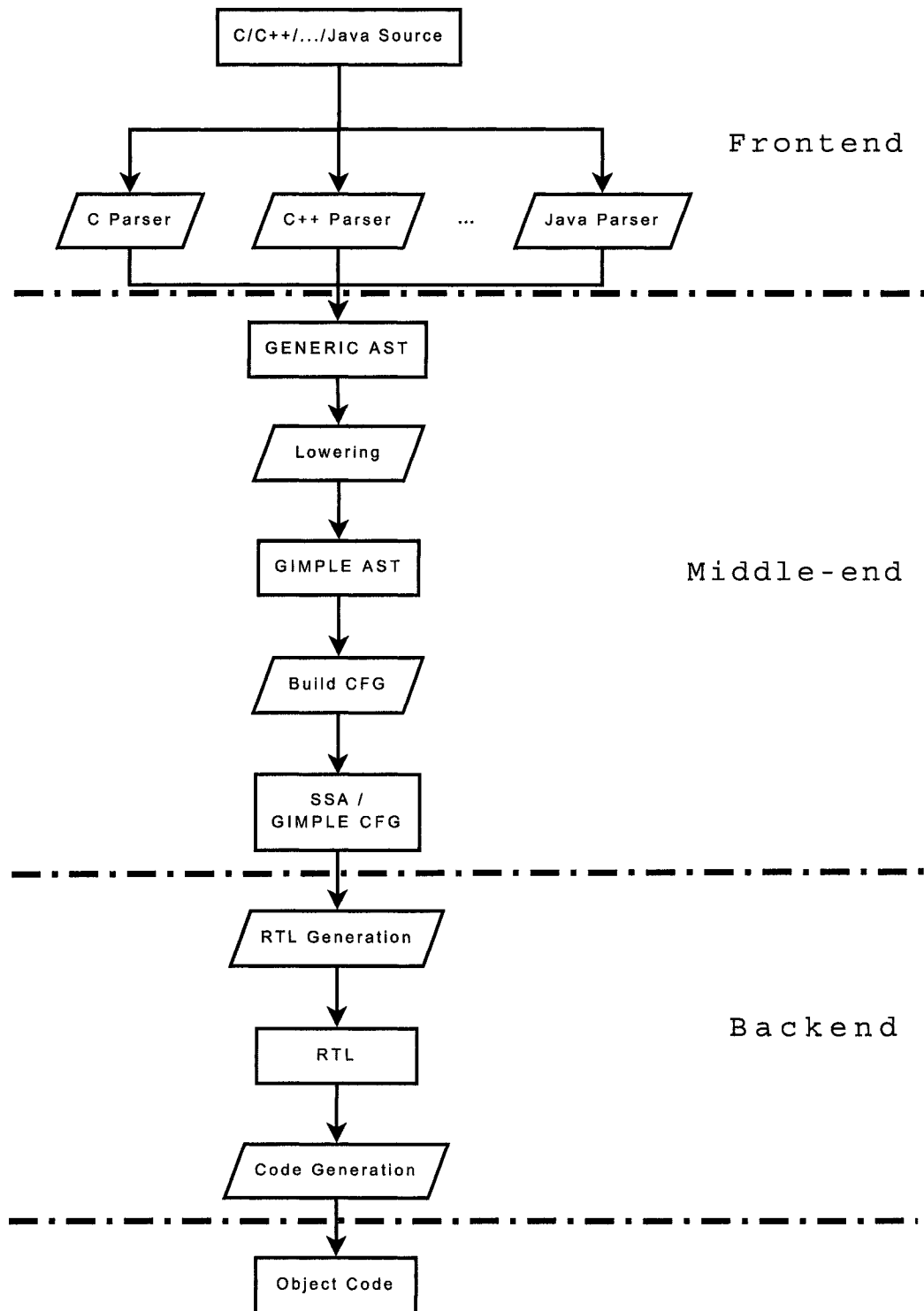


Figure 5: GCC Architecture and Compilation Phases

The GCC version 4 and above are composed of three main parts - the frontend, middle-end, and backend. The frontend accepts source code by piping it through language-specific preprocessor and parser respectively. Starting at GCC version 4, the universal output of GCC frontend is in GENERIC language. This means, all programming languages compilable by GCC share the same intermediate representation in the middle-end.

The GENERIC language contains complex statements and nested expressions, making it difficult to process. Hence, it is firstly *lowered* to a three-address code called GIMPLE in the middle-end. Temporary variables are created to decompose the nested expression. The middle-end further transforms the GIMPLE representation to GIMPLE CFG (Control Flow Graph in GIMPLE grammar), decomposing the complicated structures by introducing new goto statements and corresponding labels. Along with the above two steps of transformation, the middle-end also goes through a list of high-level optimization passes, such as alias analysis and loop unrolling.

The backend of the compiler generates RTL and object code consecutively. Details of the workflow in the backend are beyond the scope of this thesis and can be found in [42].

4.3.2 GENERIC and GIMPLE Languages

GENERIC is a language-independent intermediate language to represent all parse-tree constructs produced by GCC's various language-specific frontends. The GCC middle-end performs many passes of AST-level analysis on the GENERIC tree.

GIMPLE is a subset of GENERIC. Most CFG-level optimization passes are performed on the GIMPLE tree. Each GIMPLE tree is used to build a GIMPLE/CFG. Lexical scopes

are represented as containers. Nested expressions are decomposed to a three-address form with temporary variables storing intermediate values. Control structures such as `for` and `while` loops in C are replaced by `gotos`. Figure 6, taken from [42], shows an example GIMPLE representation and its corresponding C++ source code. A rough GIMPLE grammar is included in Appendix B

APIs are available to traverse and transform the GENERIC and GIMPLE trees. For example, the following code fragment traverse the GIMPLE/CFG tree of an entire function:

```
1  basic_block bb;
2  block_stmt_iterator iter;
3  FOR_EACH_BB(bb)
4  {
5      for (iter = bsi_start(bb);
6          !bsi_end_p(iter);
7          bsi_next(&iter))
8      {
9          tree currStmt = bsi_stmt(iter);
10     }
11 }
```

4.4 Design and Implementation

4.4.1 Adding Instrumentation Passes

Earlier, we mentioned that the GCC middle-end goes through a number of passes for code transformation and optimization. These passes are coordinated by a utility called the pass manager. Its job is to execute or conditionally skip the individual passes in the correct order, and take care of bookkeeping information that is used across the passes.

Typically, the pass manager iterates through a linked-list of registered passes. For each pass, the manager consults its *gate function* to check if the pass can be executed. If the answer is positive, the manager calls its *execute* function to run the pass.

```

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 6: Sample Source Code and Corresponding GIMPLE Representation

Our implementation starts by including two instrumentation passes in the GCC middle-end. This involves three steps:

1. Defining pass information
2. Registering our passes in the pass manager
3. Adding functionality to our passes

Defining Pass Information

Each optimization and code transformation pass in the GCC middle-end defines a structure (`tree_opt_pass`) that contains all information the pass manager needs to know about it. Our two instrumentation passes are of no exception.

```
struct tree_opt_pass pass_tree_security_instrument_vardecl =
{
  "sinstrument_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 7: Pass Information for Instrumentation Phase One

Figure 7 shows how we define the pass information for the first phase of the code instrumentation.

- The *gate* field points to the gate function, which determines whether the pass is executed or not.
- The *execute* field points to the function containing the working code of the pass.
- The *properties_required* field defines what GIMPLE tree properties are needed by the pass. For instrumentation phase one of our extension, we need a valid and language-independent GIMPLE tree (defined by enumeration value `PROP_gimple_any`).

Registering Passes

Registering a pass in the pass manager is realized by including the new passes in the `init_optimization_passes()` function defined in `passes.c`. Figure 8 shows where we inserted two new passes for code instrumentation in this function.

```

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 8: Registering Code Instrumentation Passes in `passes.c`

Adding Working Code to Passes

The working code of a pass is defined in its *execute function*. In the case of our instrumentation phase one, it corresponds to the function `tree_security_instrument_vardecl`.

4.4.2 Adding Command-line Option

Most optimization passes of the GCC middle-end can be controlled with command-line options. Adding a command-line option in GCC is a trivial task, because all options are specified in a particular source file, i.e. `common.opt`.

```
ftree-security-instrument=  
Common Report Joined Var(flag_tree_security_instrument) Init(0)  
Enable security code instrumentation
```

Figure 9: Enabling Command-Line Option for Code Instrumentation in `common.opt`

Figure 9 shows the entry of command-line option that we added in `common.opt`. A brief explanation of the entry is provided below:

- The option `ftree-security-instrument=` defines the actual option stripped of the leading hyphen.
- The second line of the option definition defines the attributes of this option. In particular, the definition `Var(flag_tree_security_instrument)` specifies that the string following the option `ftree-security-instrument=` on the command line is stored in variable `flag_tree_security_instrument` and the definition `Init(0)` initializes this variable to 0.

- The text `Enable security code instrumentation` describes this option.

It appears in the help message of the GCC compiler.

4.4.3 Recognizing Environment Variables

As previously mentioned, security analysts can enable code instrumentation in our GCC extension through defining environment variables. This is implemented by calling Linux specific `getenv()` function.

Special attention was paid to the environment variable `SHARED_LIB_NAME`, which specifies the static or shared library that is to be linked to the instrumented program. The original GCC implementation recognizes linking requirements by checking the `-l` and `-L` command-line options. All the external library names are stored in an array when GCC processes these two options (performed by function `process_command()` in `gcc.c`).

To force GCC to link with our instrumentation library defined in environment variable, we modified `process_command()` to push the library name in the array of linking library names.

4.4.4 Instrumentation Phase One - Scope-Wise Instrumentation

Table 4 shows the security-sensitive program points (where our extension can instrument code) and their corresponding instrumentation phases. In particular, instrumentations for function returns, variable declaration, and variable's exiting binding scope are performed in phase one.

As shown in Figure 8, instrumentation phase one is executed before the middle-end

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

Table 4: Instrumentation Points and Corresponding Phases

lowers the GIMPLE representation. This phase works on high-level GIMPLE representation because after lowering, all variables are moved out of their `BIND_EXPR`¹ binding context [42] and we lose liveness information of variable declarations that we intend to instrument.

Figure 10 shows the definition of `tree_security_instrument_vardecl`, the *execute function* of phase one. It calls `secinstr_xform_decls` and `secinstr_xform_out_of_scope` to instrument the code at variable declarations and end of binding scopes respectively. Both functions in turn call `walk_tree_without_duplicates` to traverse the chain of `BIND_EXPR` of a GIMPLE tree. While traversing, it searches for variables and function parameters matching the instrumentation guide and performs the instrumentation if advised.

4.4.5 Instrumentation Phase Two - CFG-Based Instrumentation

Phase two of the instrumentation targets function calls and variable uses. This is realized by traversing the GIMPLE/CFG of each function and matching the function calls and variable uses in each statement to the instrumentation guide. If a match is found, code

¹`BIND_EXPR` is a type of GIMPLE tree node for representing scopes in programs.

```

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 10: The *execute function* of Instrumentation Phase One

instrumentation is performed according to the specification in the guide.

The following paragraphs describe how the program point matching is implemented. An example of actual code instrumentation will be discussed in Section 4.4.6.

- Matching function calls.

According to GIMPLE grammar (see Appendix B), function calls are represented by *CALL_EXPR* tree nodes. The *CALL_EXPR* tree node can appear only in two places on a GIMPLE tree: It may be a statement by itself, if it does not have side effect; it may be the right operand of an assignment statement (represented by a *MODIFY_EXPR* tree node). Accordingly, our implementation searches each statement and its right operand (if there is one) for *CALL_EXPR*.

Once a *CALL_EXPR* tree node is found, we use the API:

```
lang_hooks.decl_printable_name (CALL_EXPR_typed_tree, 2)
```

to retrieve the name of the callee function and compare it to those in the instrumentation guide.

- Matching variable read and write.

Matching variable read and write is fairly easy. Since all GIMPLE trees are in three-address form, we are sure that only the left operand of an assignment statement can be written and the variables used in the right operands are only read. Therefore, we passed a read-write flag to our variable searching routine to differentiate variable uses.

- Matching pointer dereferences.

Pointer dereferencing is represented by *INDIRECT_REF* tree node. The GIMPLE

grammar allows `INDIRECT_REF` trees to be used as both left and right operands of an assignment statement. However, we decided to pass read-write flag to our search routine as well. This flag is currently neglected. If future extension to our implementation requires to differentiate read and write access of dereferenced pointers, this flag can be used for that purpose.

4.4.6 Instrumentation Example - Supporting Suppression

Function	Purpose
GIMPLE/CFG API:	
<code>split_block(basic_block, block_stmt_iterator)</code>	split a basic block into two before the given iterator
<code>make_edge(basic_block, basic_block, EDGE_TYPE)</code>	make an edge of <code>EDGE_TYPE</code> between the given blocks
<code>find_edge(basic_block, basic_block)</code>	find the edge connecting to blocks
<code>set_immediate_dominator(DOMINATOR_TYPE, basic_block, basic_block)</code>	set the first block to be the dominator of the second one
TREE CONSTRUCTION API:	
<code>create_tmp_var(tree type, const char* name)</code>	create a tree node of a temporary variable of the given type and name
<code>build #(...)</code>	create a tree node with # number of operands
<code>bsi_insert_after(block_stmt_iterator*, tree, INSERT_MODE)</code>	insert a tree after the given iterator for the given mode
<code>SET_EXPR_LOCUS(tree, location_t*)</code>	set the source location of a tree

Table 5: GIMPLE APIs Used for Instrumenting Suppression Code

Implementing code instrumentation that can suppress certain statements in the original program is not so straightforward that it involves a series of tree transformation. A number of GIMPLE/CFG APIs in Table 5 are used to accomplish our goal.

Figure 11 illustrates the two-step transformation of a GIMPLE tree to instrument the

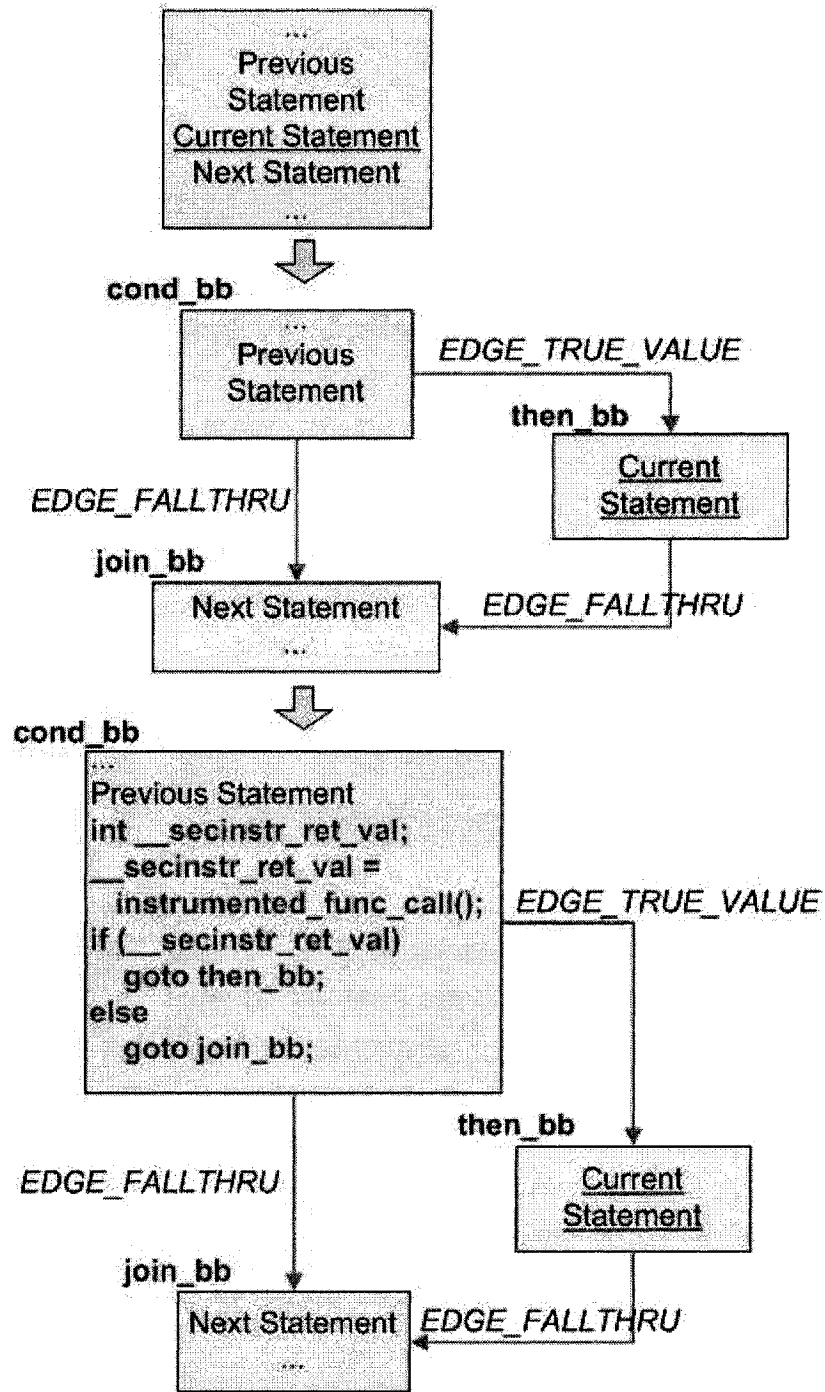


Figure 11: Instrumentation of Code Suppression

code suppression in the program. In this illustration, our goal is to instrument a call to `instrumented_func_call()` before "*Current Statement*" and suppress it if the instrumented function returns 0.

The first step splits the basic block containing "*Current Statement*" into three blocks, i.e. *cond_bb*, *then_bb*, and *join_bb*; then creates edges to connect these blocks. Note that edges of GIMPLE/CFG are typed. For example, if a condition expression is evaluated true at the end of *cond_bb*, execution will follow `EDGE_TRUE_VALUE`-typed edge to *then_bb*.

The second step instruments the call to `instrumented_func_call()` and stores its return value into a temporary variable. A branching statement is then appended to the end of *cond_bb* to evaluate the return value.

4.5 Summary

In this chapter, we introduced the implementation of a new approach to code instrumentation - extending GCC with instrumentation functionality. The motivation of this research thread was to develop a software utility for injecting program monitoring routines into a testee program, so that when it executes, the program monitor can dynamically detect violations of security properties.

We decided to choose this approach based on two reasons. Firstly, compiler-aided code instrumentation is more efficient than source-to-source code transformation approaches and more powerful than pre-processor-assisted and binary wrapping approaches. Secondly, extending GCC facilitates integration of code instrumentation to the automatic build system of many FOSS projects.

Our prototype implementation adds two interfaces to the GCC compiler such that code instrumentation can be enabled either through command-line option or through environment variable definition. The extension accepts an instrumentation guide to inject code at a selected set of program points. The grammar of the input guide was briefly introduced in this chapter.

Chapter 5

Integrated System and Experiments

5.1 Integrated System

This section introduces the interface, functionality and implementation of our integrated system for dynamic vulnerability detection.

5.1.1 System Overview

Figure 12 illustrates the overview of the user interface of our integrated system. Area 1 (square-framed 1 in the figure) displays all the FOSS projects that are created in our system. That being said, our system can manage and analyze multiple software projects. Area 3 displays the detected vulnerabilities of a project. Area 2 is the main view of the system. From left to right in this area, four tabs present different views and functionalities of our system. Respectively, they are:

- **Project Overview** shows the general description of the currently selected project.

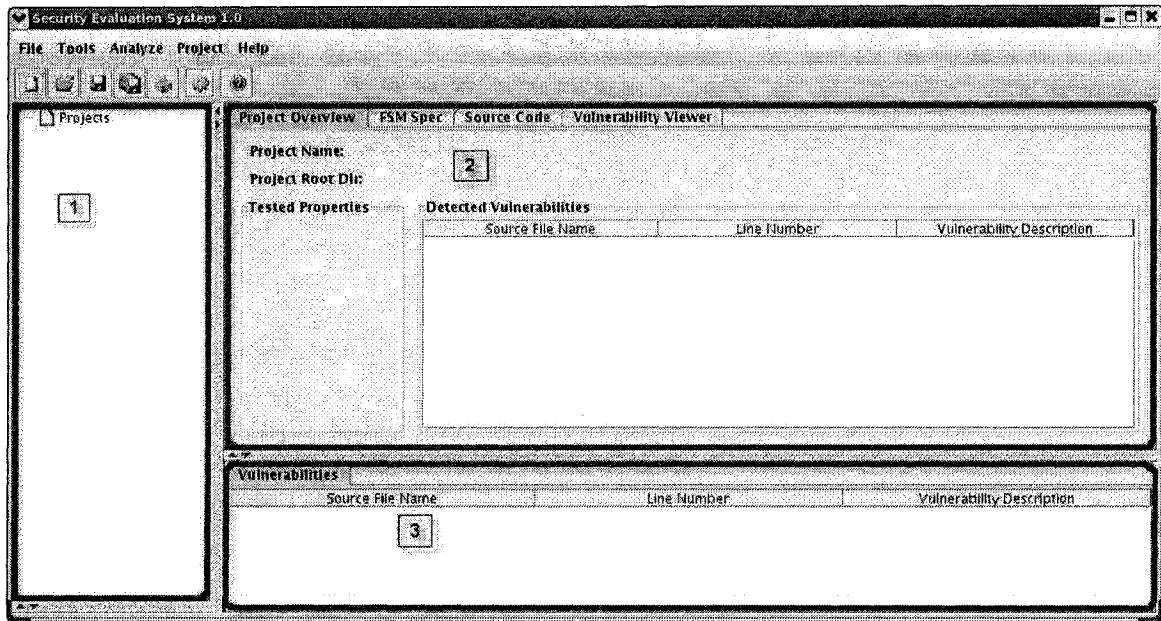


Figure 12: Integrated System - Overview

- **FSM Spec** is a GUI editor for security analysts to graphically compose security properties.
- **Source Code** is a GUI text editor with syntax highlighting, where security analysts can edit the source code to fix revealed bugs.
- **Vulnerability Viewer** hosts a source code navigator, which displays the file name, line number, and source code corresponding to particular vulnerabilities.

5.1.2 System Configuration

At the first time when our system is run, users are prompted with a dialog that asks them for some system-specific settings. Figure 13 shows these settings:

- **secgcc Bin Directory** is the directory containing the executable file of our GCC extension.

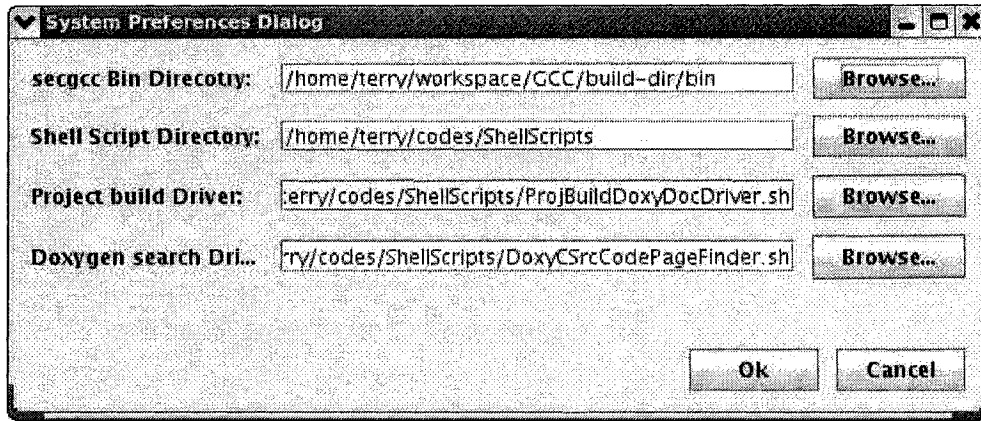


Figure 13: Integrated System - Preference Dialog

- **Shell Script Directory** is the directory containing the shell scripts that drive the execution of project building and analysis tasks.
- **Project build Driver** is a specific shell script that builds a FOSS project and generates Doxygen [3] documentation of it. Our current implementation includes a template shell script that assumes Make is the automatic build system used by the project. Users are free to modify it or supply their own driver script in order to support other automatic build systems such as Ant.
- **Doxygen search Driver** is an optional shell script that helps searching Doxygen documentation for particular program points. Details are covered in Section 5.1.6.

During later executions, users are also free to modify the system configuration by invoking the configuration dialog through a program menu as shown in Figure 14. The design decision of allowing configurable choices of the compiler and various shell script is to facilitate future extension of the system. As our system is only the first phase a larger on-going security analysis tool, more features such as static analysis and support of more programming languages such as C++ and Java are expected to be added to it. Accordingly, it will be

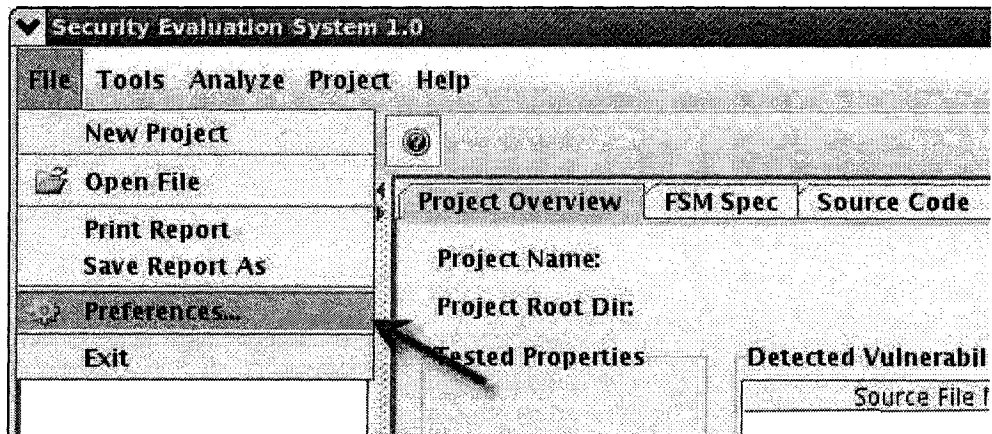


Figure 14: Integrated System - Configure System Settings through *File* Menu

easier to extend the capability of our system through the modification of these settings.

5.1.3 Project Management

Project Creation

Our system offers basic project management functionalities, i.e. project creation, configuration, building, execution, and vulnerability reporting. All these functionalities are accessible through the *Project* menu as shown in Figure 15.

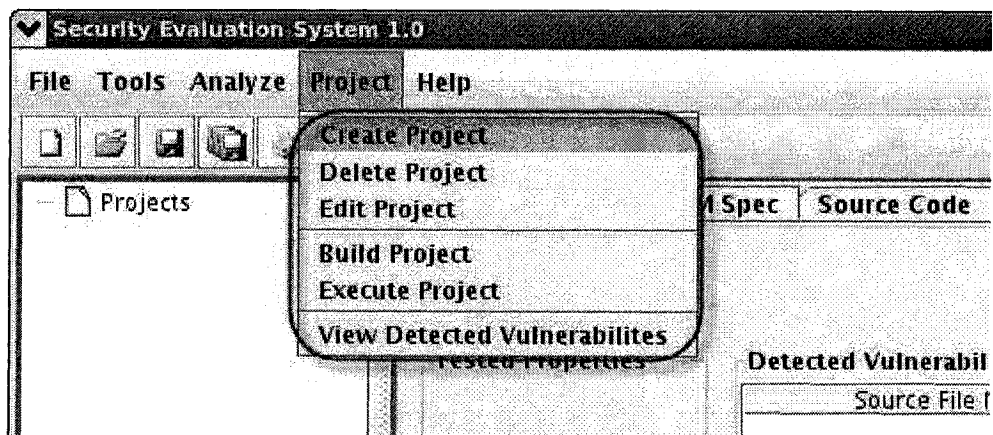


Figure 15: Integrated System - Project Management through *Project* Menu

In order for our system to perform dynamic vulnerability detection, security analysts

need to create a project corresponding to the FOSS project they intend to evaluate. This is achieved through menu *Project*→*Create Project*. A dialog as shown in Figure 16 then prompts up and asks for the description and the source directory of the newly added FOSS project.

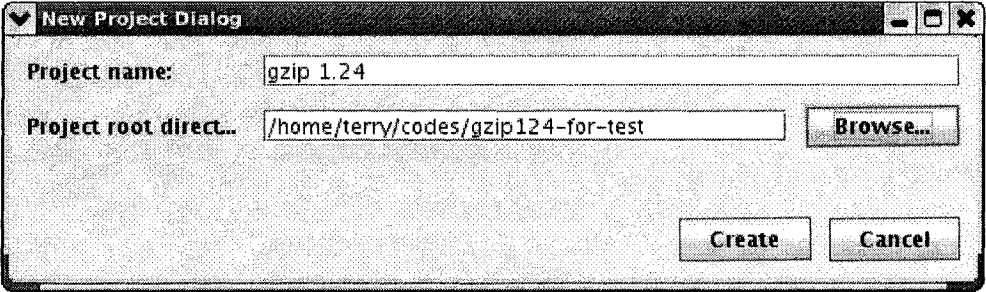


Figure 16: Integrated System - Project Creation Dialog

Once the *create* button in the dialog is clicked, our system registers the newly added project, searches for all its source codes, and displays these information in the system's project list. This is illustrated in Figure 17.

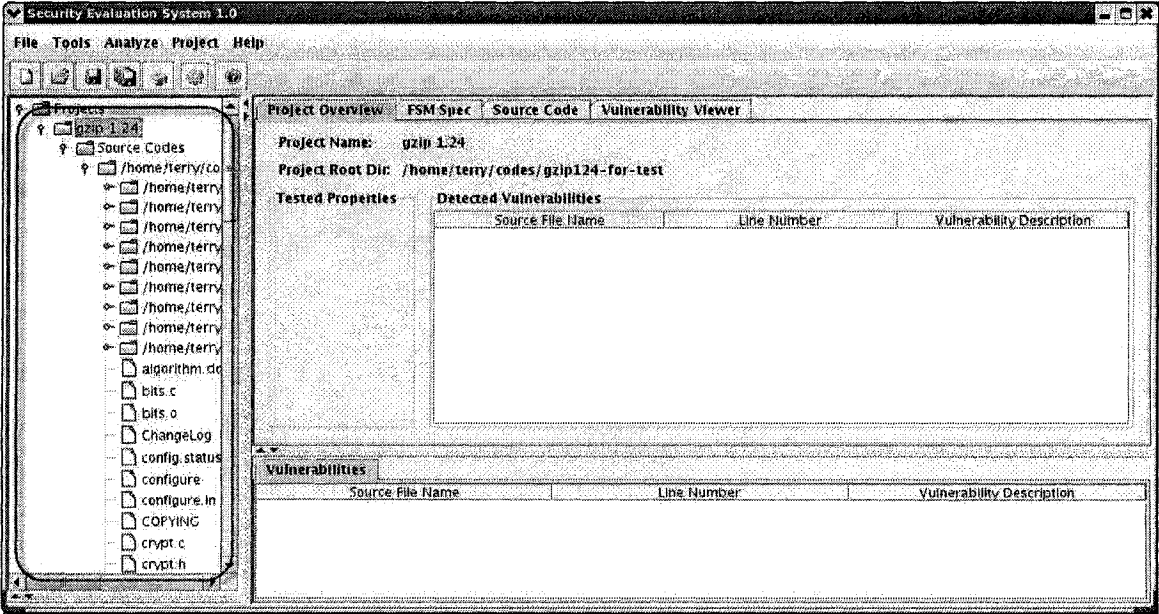


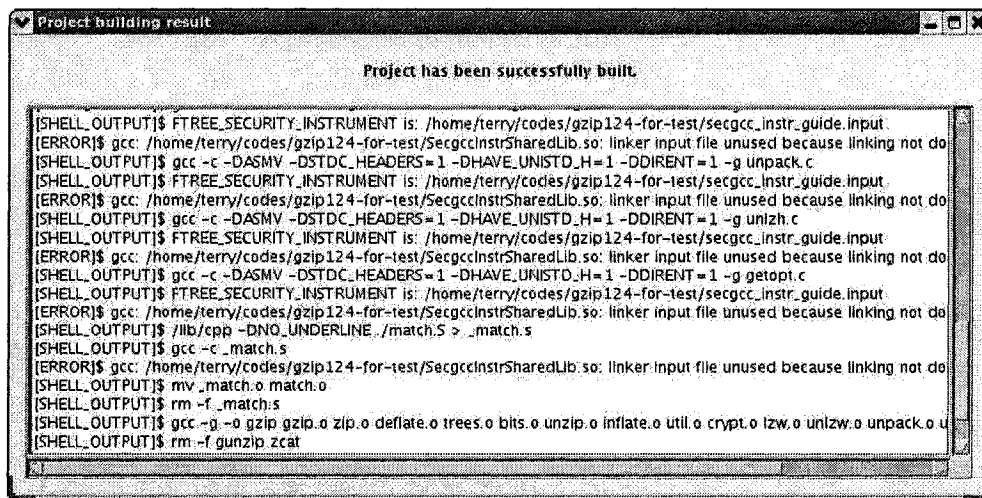
Figure 17: Integrated System - Newly Created Project in Project List

Security analysts then define the security properties they intend to check against the project or reuse some of the existing pre-defined properties. The details of property specification is deferred to later paragraphs.

Project Building

Our system can build a project with or without security properties being specified. In the first case, the system simply performs a normal build of the FOSS project. In the latter case, code instrumentation is performed during compilation.

Our current implementation uses a shell script to automate project building. It assumes that a `Makefile` locates in the root source code directory of the testee program. This is true for most FOSS projects. If there is a conflict with this assumption, security analysts can always modify the behavior of our system through re-configuring the system or modifying the template script (as previously mentioned).



```
Project building result
Project has been successfully built.
[SHHELL_OUTPUT]$ FTREE_SECURITY_INSTRUMENT is: /home/terry/codes/gzip124-for-test/secgcc_instr_guide.input
[ERROR]$ gcc: /home/terry/codes/gzip124-for-test/SecgccInstrSharedLib.so: linker input file unused because linking not do
[SHHELL_OUTPUT]$ gcc -c -DASMV -DSTDC_HEADERS=1 -DHAVE_UNISTD_H=1 -DDIRENT=1 -g unzip.c
[SHHELL_OUTPUT]$ FTREE_SECURITY_INSTRUMENT is: /home/terry/codes/gzip124-for-test/secgcc_instr_guide.input
[ERROR]$ gcc: /home/terry/codes/gzip124-for-test/SecgccInstrSharedLib.so: linker input file unused because linking not do
[SHHELL_OUTPUT]$ gcc -c -DASMV -DSTDC_HEADERS=1 -DHAVE_UNISTD_H=1 -DDIRENT=1 -g unizh.c
[SHHELL_OUTPUT]$ FTREE_SECURITY_INSTRUMENT is: /home/terry/codes/gzip124-for-test/secgcc_instr_guide.input
[ERROR]$ gcc: /home/terry/codes/gzip124-for-test/SecgccInstrSharedLib.so: linker input file unused because linking not do
[SHHELL_OUTPUT]$ gcc -c -DASMV -DSTDC_HEADERS=1 -DHAVE_UNISTD_H=1 -DDIRENT=1 -g getopt.c
[SHHELL_OUTPUT]$ FTREE_SECURITY_INSTRUMENT is: /home/terry/codes/gzip124-for-test/secgcc_instr_guide.input
[ERROR]$ gcc: /home/terry/codes/gzip124-for-test/SecgccInstrSharedLib.so: linker input file unused because linking not do
[SHHELL_OUTPUT]$ /lib/cpp -DNO_UNDERLINE ./match.S > ./match.s
[SHHELL_OUTPUT]$ gcc -c ./match.s
[ERROR]$ gcc: /home/terry/codes/gzip124-for-test/SecgccInstrSharedLib.so: linker input file unused because linking not do
[SHHELL_OUTPUT]$ mv ./match.o match.o
[SHHELL_OUTPUT]$ rm -f ./match.s
[SHHELL_OUTPUT]$ gcc -g -o gzip gzip.o zip.o deflate.o trees.o bits.o unzip.o inflate.o util.o crypto.o lzw.o unizw.o unpack.o u
[SHHELL_OUTPUT]$ rm -f gunzip.zcat
```

Figure 18: Integrated System - Dialog of Project Building Result

The result of project building is displayed in a dialog as in Figure 18. From the dialog, security analysts can access the complete console messages that are emitted during

compilation, which is helpful to diagnose any compilation errors.

Project Execution

Once a project is built, security analysts can execute it by selecting the executable file and defining command-line options, as illustrated in Figure 19.

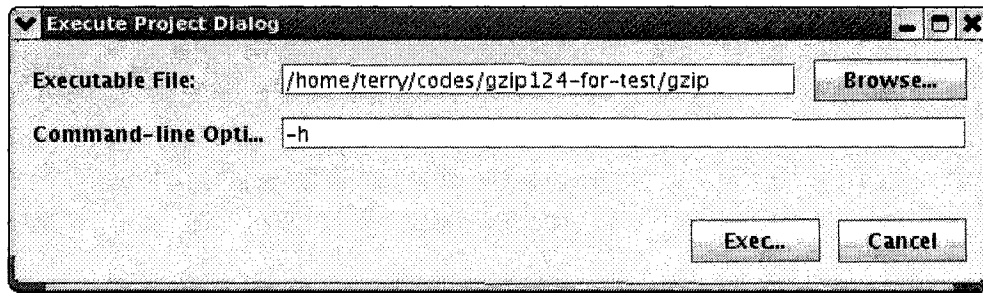


Figure 19: Integrated System - Dialog of Project Execution Configuration

Our current implementation enables security analysts to manually generate test cases for a FOSS project then execute it through the interface shown in Figure 19. The experiments that we will show in Section 5.2 were conducted through this interface. We expect that an automatic test case generator and a program execution driver will be developed to replace the current interface in future extension.

5.1.4 Vulnerability Report, View and Fix

When executing an instrumented executable of the testee program, violations of security properties are detected and recorded. Security analysts can opt for displaying the complete list of all the detected property violations, as shown in Figure 20. For each detected property violation, the system provides the exact program point (identified by the source file name and line number of the concerned statement) where the violation is detected to help

security analysts debug the program. It also prints a brief description of the violation. For example, Figure 20 shows that at line 1725 of gzip.c of the tested project “gzip 1.24”, a double-freeing vulnerability is detected.

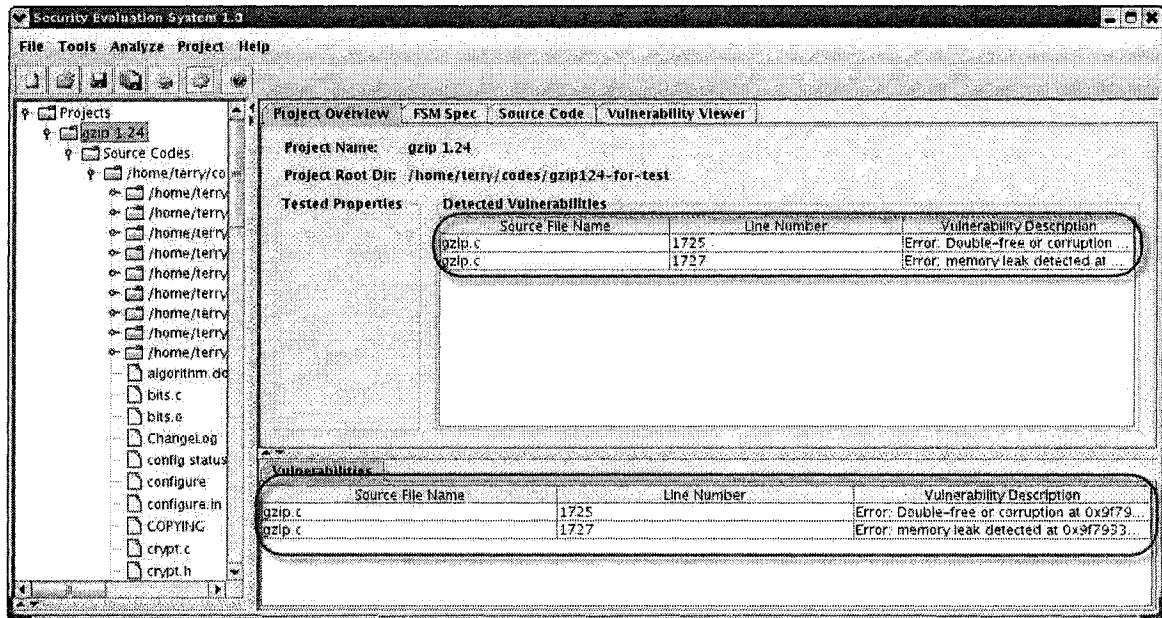


Figure 20: Integrated System - List of Property Violations

A click on any particular vulnerability list entry leads security analysts directly to the detailed vulnerability view, where the vulnerable statement in the source code is displayed and high-lighted. This is illustrated in Figure 21.

In addition to simply displaying the detected security vulnerabilities, our system also allows analysts to fix and re-test them. Particularly, we included a multi-tabbed text editor (see Figure 22) with syntax highlighting for multiple programming languages.

5.1.5 Security Property Specification

In Section 3.3, we mentioned that the state transitions in component edit automata are specified with the help of State Machine Compiler. We also introduced the implementation

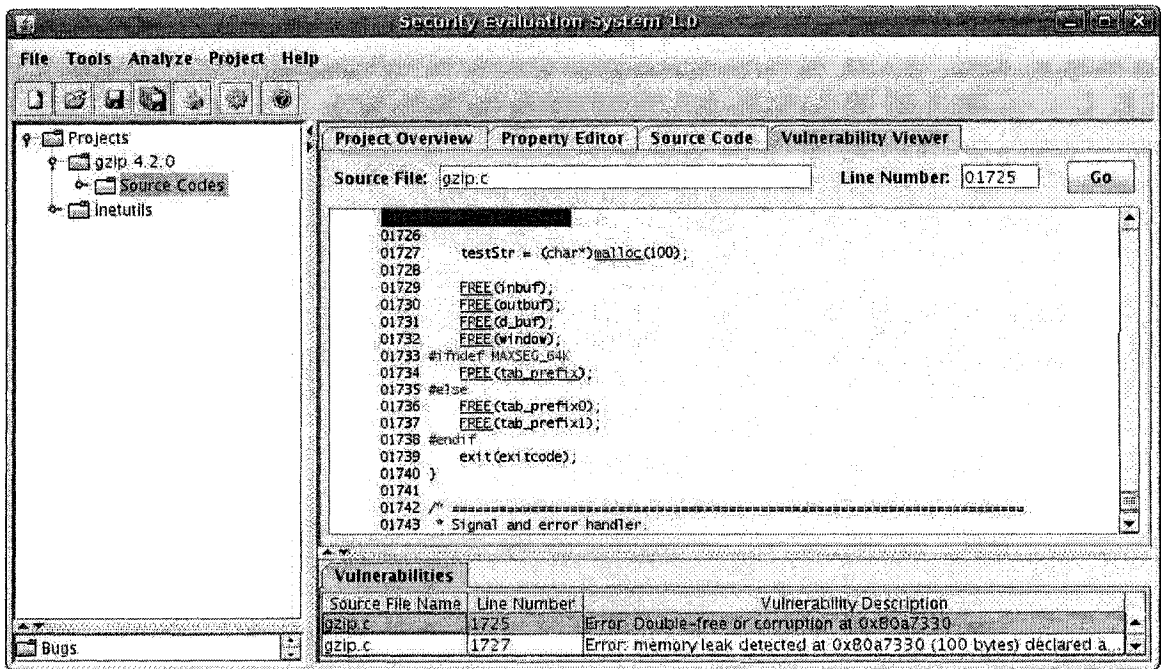


Figure 21: Integrated System - Detailed Vulnerability View

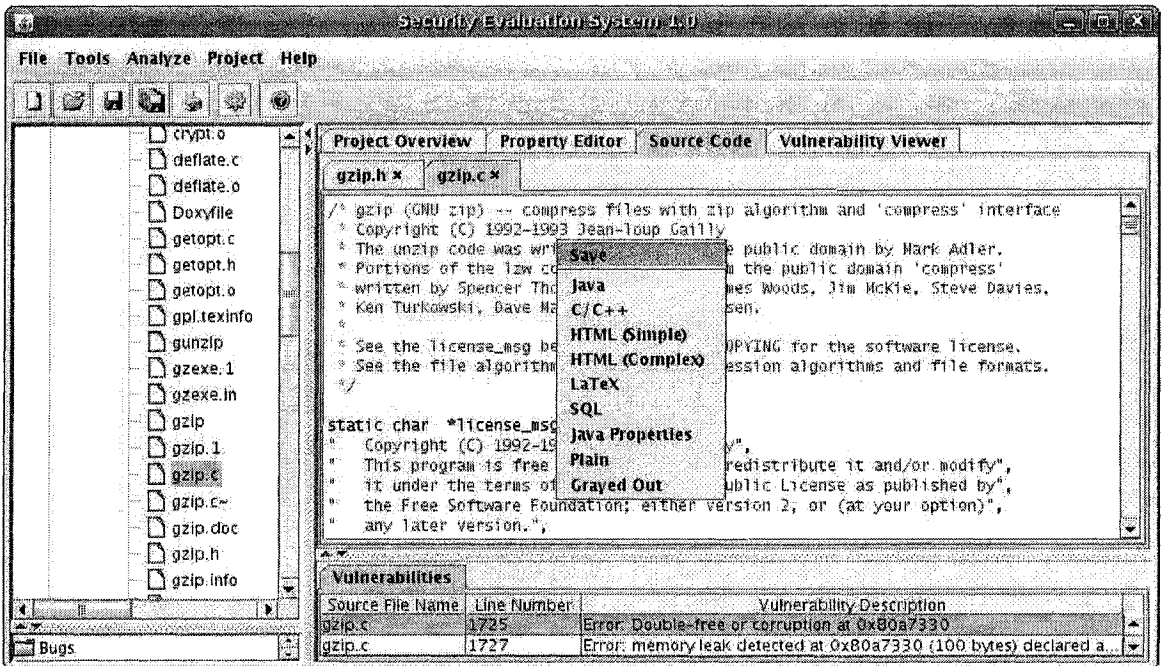


Figure 22: Integrated System - Source Code Editor for Bug Fixing

of *Team Edit Automata* as a hierarchy of classes so that security properties can be written in a programmer-friendly manner. Experienced security analysts can certainly use a text editor to accomplish such task. However, programming from scratch in a text editor is a tedious and error prone practice. Moreover, the written property specifications are not easy to read and understand. Therefore, our system provides a graphical interface to let the analysts visually specify security properties. The system automatically converts the graphic property specification to corresponding implementation code.

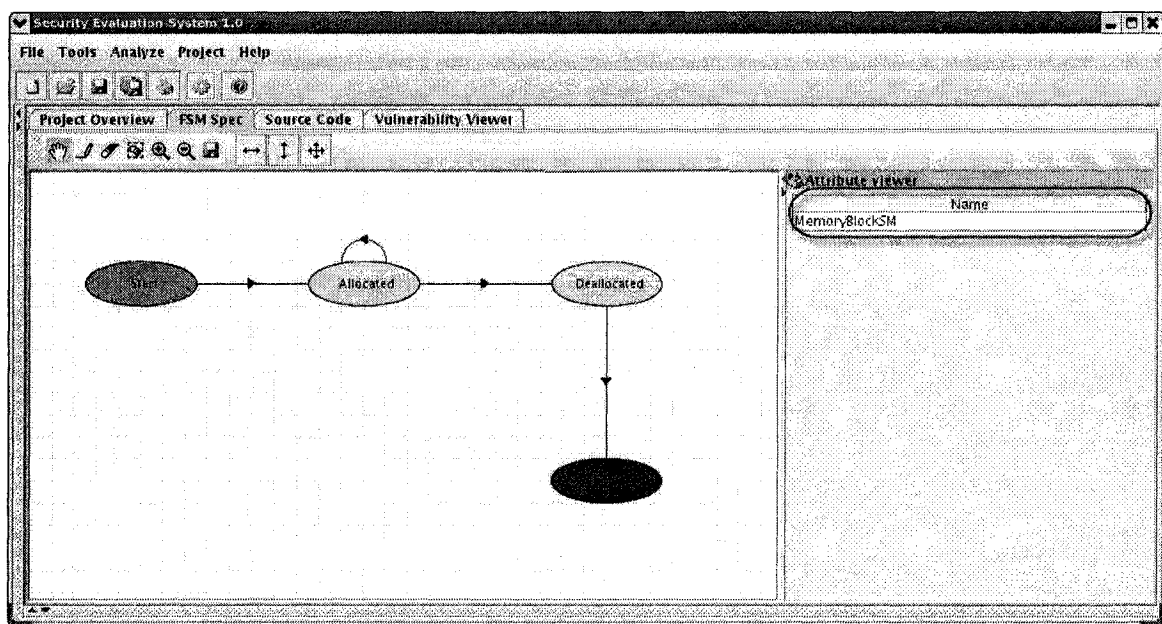








Figure 23: Integrated System - Overview of Security Property Editor

Figure 23 shows the overview of the Security Property Editor in our system. The toolbar offers access to most of the graph editing functionalities. The left-most three icons allow the analysts to change the graph edit mode, i.e.  for *selection* mode,  for *edit* mode, and  for *deletion* mode. The right-most three icons (  ) allow the analysts to change the display mode of graph, i.e. fitting canvas width, canvas height, and canvas size respectively from left to right. The buttons in the middle allow security analysts to zoom

in/out the graph and save it. The icons of these buttons are standard and self-explaining.

Figure 23 shows that the analysts can specify the name of the property in the *Attribute viewer* (annotated with a round-cornered rectangle in the figure). In addition to the name, the analysts generally specify much more information of a security property, including adding states and transitions in the graphic editor and defining their attributes.

Adding a State

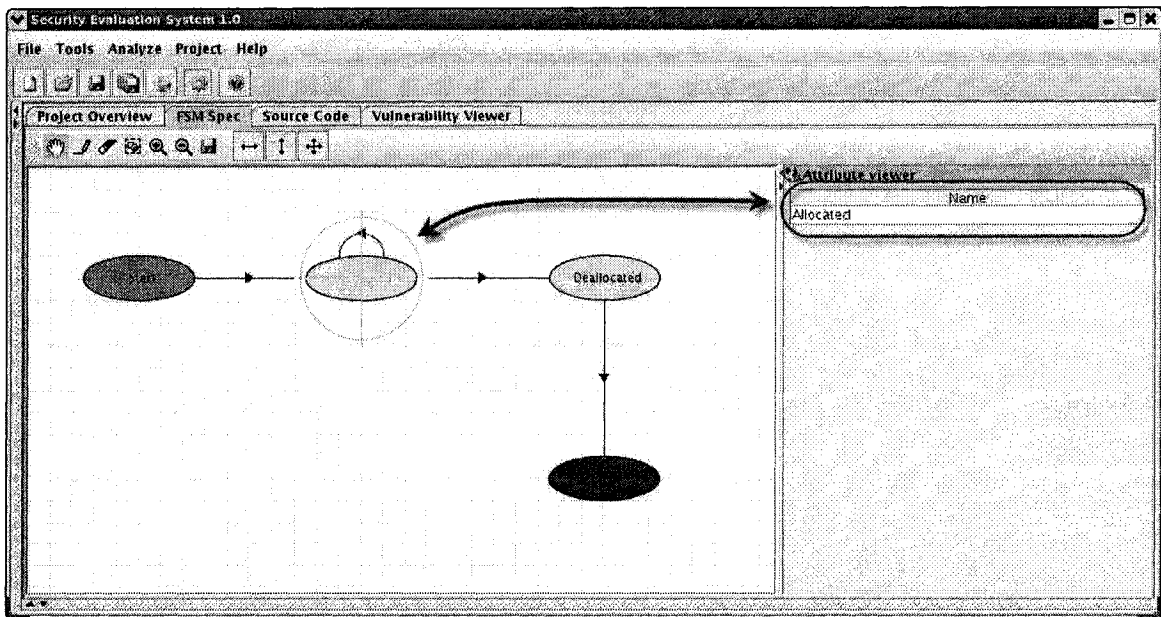


Figure 24: Integrated System - Specifying State Name in Security Property Editor

Adding a state in the security property editor involves two steps. Firstly, the analysts click the toolbar button to switch to *edit* mode and then click anywhere on the canvas to create a new state node. A default name is automatically generated for the new node. Secondly, they optionally provide a more meaningful name to the new state in the *Attribute viewer*, as shown in Figure 24.

Adding a Transition

In *edit* mode, the analysts can add a transition in three ways. Firstly, if both start and destination states are present on the canvas, a new transition can be added by mouse-clicking the start state than releasing in the destination state. Secondly, if only the start state is present, by mouse-clicking the state and releasing in any empty space on the canvas, a new transition and the new destination state are created. Thirdly, if the start and destination states are the same state on the canvas, a new *loop-back* transition can be added by mouse-clicking the state than dragging and releasing in the same state.

In reality, for the same pair of start and destination states, there might exist multiple transitions (e.g. different input actions may trigger identical state change with different output actions). Accordingly, our system allows the analysts to add multiple transitions between two states.

Defining Transition Event

A transition is properly defined by an event, an optional guard condition, a list of actions as well as its start and destination states. Figure 25 shows how to define the transition event in the security property editor.

Particularly, the analysts need to give a name to the event, and optionally specify a list of event arguments. In the current implementation, the analysts are free to use any primitive data type in the C language for these arguments.

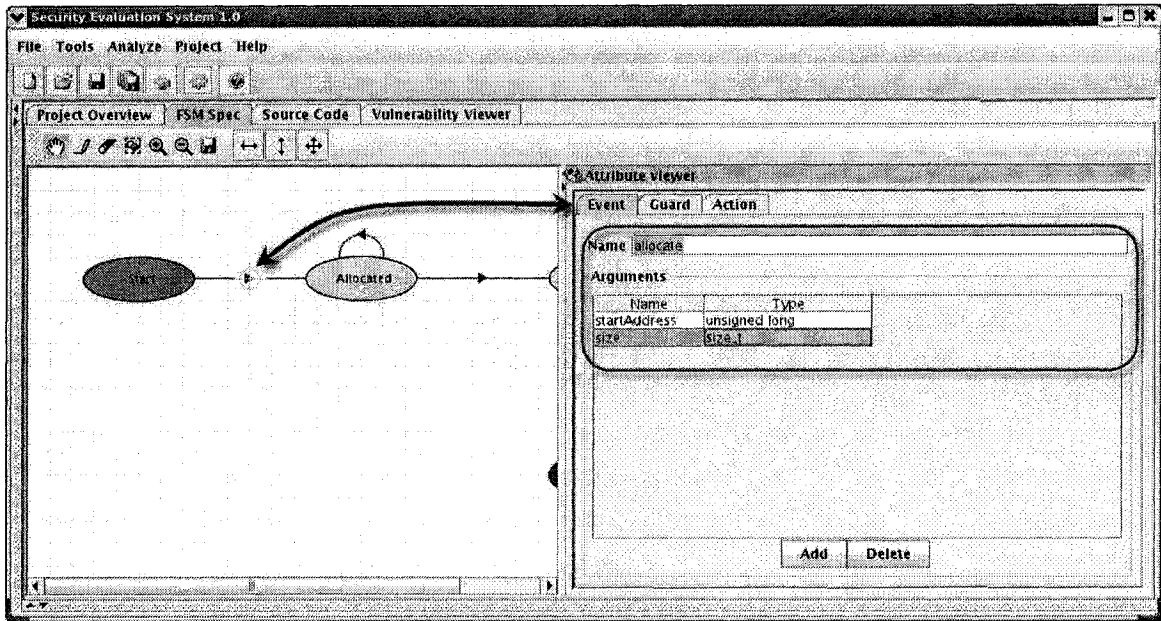


Figure 25: Integrated System - Defining Transition Event in Security Property Editor

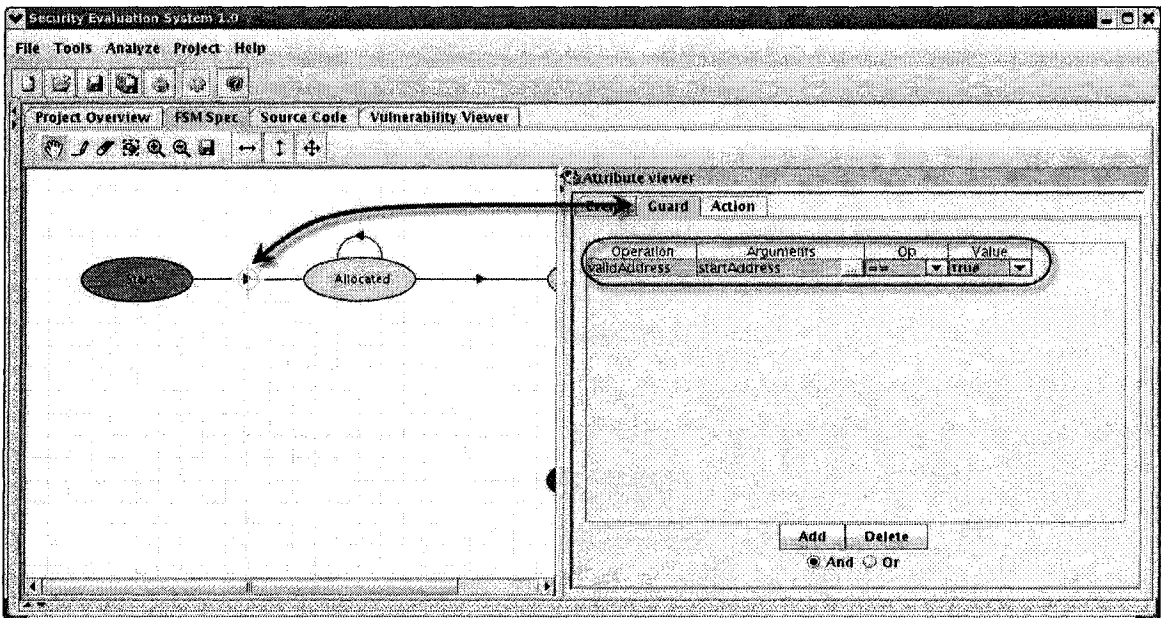


Figure 26: Integrated System - Defining Transition Guard in Security Property Editor

Defining Transition Guard

The transition guard of our component edit automata is implemented as conjunction and/or disjunction of logical expressions. The security property editor of our system allows the analysts to write the guard condition in a visual manner. Figure 26 shows the attribute editor for transition guard.

Particularly, the analysts can add zero or more simple logic expressions to the guard condition. The expressions are added either as conjunction or disjunction to the existing ones.

Defining Transition Actions

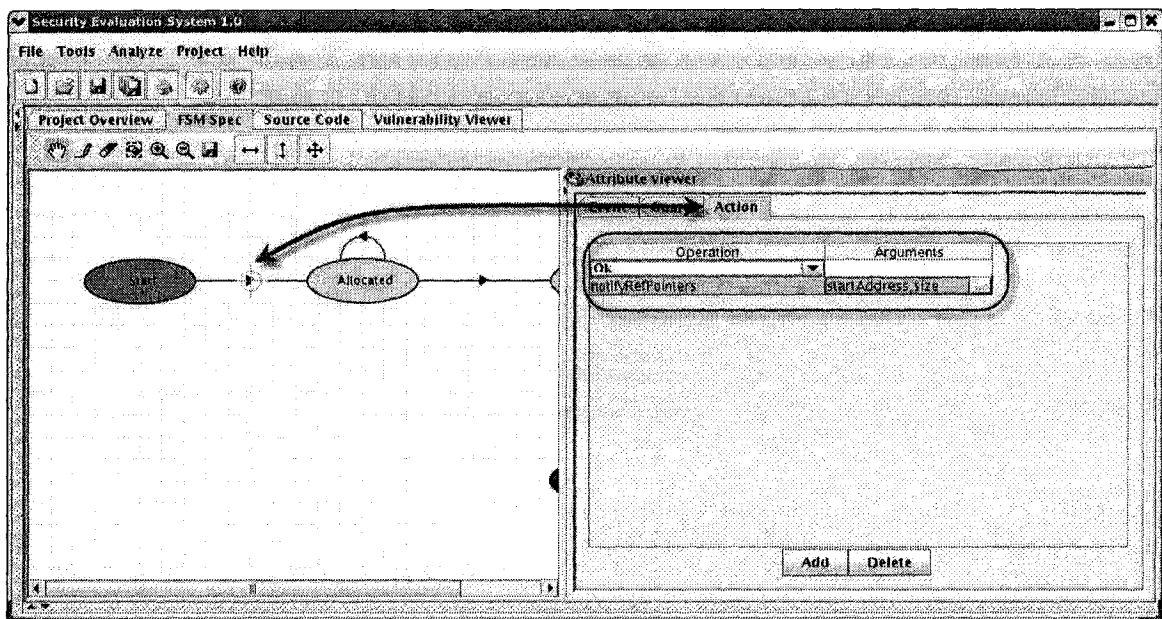



Figure 27: Integrated System - Defining Transition Actions in Security Property Editor

Event transition of our component edit automata must at least emit one action, i.e. one of Ok, suppress, insert, or halt. This is reflected in the graphic property editor as a compulsory action in the *Action* tab of the *Attribute Viewer*. The analysts may select any

one of these four action from the drop-down list (see Figure 27).

Additional actions can be defined on the transition, as shown in Figure 27. When the analysts save the property, our system automatically generates the prototype and an empty implementation body for each additional action, where the analysts can insert functioning code.

Saving Security Properties

Once the analysts finish editing a security property in the graphic editor of our system, they click the  button to save it. This triggers a series of tasks that our system performs without the analysts' knowledge.

1. The graphic components on the canvas are serialized to an XML file, so that next time when the property is loaded to the security property editor, these components and their attributes are correctly restored and displayed.
2. A .sm file is generated corresponding to the state transition defined by the property specification.
3. The State Machine Compiler is called on the .sm file to automatically generate the C++ code of a finite state machine corresponding to the property's state transition definition.
4. The context class of the finite state machine is automatically generated as a subclass of `ComponentAutomata`.

When all the above tasks are accomplished, security analysts only needs to insert the functioning code to the context class to finalize the property specification. If security analysts choose to check the property at later stage, the implementation code is compiled into a shared library and instrumented to the testee program.

5.1.6 System Implementation

GUI Implementation

The graphic interface of our integrated system is developed completely in Java and compatible with Java 5.0 and above.

Two external contributions to the GUI implementation deserve credits:

- The graphic security property editor is based on the Graph library of Real Time Studio developed by Dr. Rachid Hadjidj [48].
- The multi-language syntax highlighting text editor uses the free syntax highlighting library from Mr. Stephen Ostermiller [69].

Implementation of Automatic Project Building

Earlier we mentioned that our system can automatically build and instrument a FOSS project with the only knowledge of where the project's source code locates. This automation is carried out by a shell script. Particularly, the script performs the following tasks:

1. Setting up project specific environment variables;
2. Identifying the compiler and Makefile to be used with the project;

3. Building the project optionally with code instrumentation;
4. Instantiating a Doxyfile from the Doxyfile temple and then generating the Doxygen documentation of the entire project.

The Doxygen documentation of the project is used by our system to facilitate vulnerability report.

5.2 Experiments with the Integrated System

5.2.1 Experiment Environment

All our experiments of the integrated system were conducted with the following system setups:

- **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_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

5.2.2 Experiment 1: Checking Memory Management Vulnerabilities

Experiment Objective

This experiment aimed at proving the effectiveness of our solution, i.e. dynamically checking user-defined security property can be achieved by monitoring program execution with *Team Edit Automata*.

We chose to detect memory management vulnerabilities, because they persist in C programs. Moreover, many existing static and dynamic security analysis tools check these vulnerabilities, as we mentioned in Section 2.2. Therefore, we were able to compare the performance of our system with that of existing tools.

Experiment Methods

The security property for safe memory management and access is specified in the *Security Property Editor* of our integrated system, which in turn generates *MemorySM.sm* in the syntax of State Machine Compiler (SMC). Appendix C.1.1 includes the complete code of *MemorySM.sm*.

The state of each allocated memory blocks (both on the stack and heap) is captured by one `TeamAutomata` object. The `TeamAutomata` object consists of one `ComponentAutomata` subclass object that composes a state machine object generated from *MemorySM.sm* and multiple `ComponentAutomata` subclasses to capture the states of the associated variables.

The specification encompasses the following vulnerabilities related to memory management:

- Dereferencing wild and null pointers
- Freeing wild and null pointers
- Double-freeing
- Memory leak
- Out-of-boundary memory access
- Out-of-boundary array indexing
- Reading uninitialized memory

We created a list of benchmark C programs (see Appendix C.2), which are known to have elusive memory management vulnerabilities. We used our integrated system and two well-known security analysis tools to check memory management vulnerabilities in these programs. When checking programs with our system, we manually generated test cases to drive program execution. We will discuss the limitation of our current system further in Chapter 6.

Experiment Results

Table 6 shows the experiment results. We summarize these results as following:

- Checking Program 1:

Program 1 has an out-of-boundary array indexing vulnerability, which is triggered when the index variable is modified before accessing the array. Among the three tools we used to check the program, two (including our system) successfully detected

Vulnerable Programs		Experiment Results		
Program Name	Vulnerability Description	Tools	Result Message	Notes
Program 1	Out-of-boundary array indexing	Our System	Error: Illegal array indexing with index value -2 at Experiment1.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 2	Double-freeing	Our System	Error: Double-free or corruption at 0x80a68b0 at Experiment2.c::17	
		Klocwork K7.5	PASSED: 0 Errors, 0 Warnings, 0 Filtered	
		Insure++ 5.1	Freeing dangling pointer, 1 unique occurrence 1 at an unknown location	Debug assertion failed and program crashed.
Program 3	Buffer-overflow	Our System	Error: Illegal memory access at 0x80a7990 at Experiment3.c::20	
		Klocwork K7.5	PASSED: 0 Errors, 0 Warnings, 0 Filtered	
		Insure++ 5.1	INSURE_ERROR: Internal error, 1 unique occurrence	
Program 4	Buffer-overflow	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	couldn't be compiled with Microsoft Visual C++ 6.0 & Insure++ 5.1	
Program 5	No Exploitable Vulnerability	Our System	(no error was reported)	
		Klocwork K7.5	Experiment5.c(25):Severe:Double freeing of freed memory pointed by 'buf2'. NOT PASSED: 1 Errors, 0 Warnings, 0 Filtered	
		Insure++ 5.1	couldn't be compiled with Microsoft Visual C++ 6.0 & 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 6: Experiment Result of Checking Memory Management Vulnerabilities

the vulnerability. Klocwork failed in this case. We assumed that the static analysis performed by Klocwork is not sophisticated enough to consider the data flow that affects the indexing variable.

- Checking Program 2:

Program 2 has a double-freeing vulnerability, resulting from freeing one of the two circular-referencing pointers. Our system successfully detected the double-freeing action and avoided program crash by suppressing the second freeing action. Insure++ was able to detect the vulnerability as well. However, the testee program crashed after a debug assertion failure was reported. Klocwork failed to detect the vulnerability in Program 2.

- Checking Program 3:

Program 3 is vulnerable to buffer overflow, where the buffer size is dynamically determined by user input and an unsafe string manipulation function was used without checking buffer size. Our system detected an illegal memory access and reported the vulnerable statement in the program. Insure++ also detected an error, but failed to identify the cause. Klocwork did not report any detected vulnerability or error.

- Checking Program 4:

Program 4 has a similar vulnerability as Program 3. Our system detected buffer overflow as an illegal memory access. Klocwork did not report any error. As for Insure++, we were not able to compile or instrument the program in the Microsoft Visual C++ 6.0 (VC++-6.0) IDE bundled with Insure++. This is because our program is written in C and VC++-6.0 uses a C++ compiler.

- **Checking Program 5:**

There is no exploitable vulnerability in this program. The freeing action at line 22 will never execute and so double-freeing will never occur. We executed this program in our system with integers from 1 to 10 without detecting any vulnerability. Klocwork detected a double-freeing vulnerability at line 25. This is certainly a false positive. Again, we could not compile the program with VC++-6.0 and Insure++.

- **Checking Program 6:**

Program 6 has an buffer overflow vulnerability associated with a static array. The vulnerability roots in the unsafe string printing function provided by standard C library. Our system detected an illegal memory access where the string printing function is called. Insure++ detected an unknown error and Klocwork failed to find any error.

To sum up, the above experiment results demonstrated that dynamically monitoring and checking program actions against our security properties can effectively reveal security properties. Some of these vulnerabilities are not easy to detect with the static analysis approach.

5.2.3 Experiment 2: Scalability and Usability Test

Experiment Objective

Scalability and usability are both crucial to our system, because the objective of our ongoing research project is to devise an automatic security testing tool. Accordingly, our integrated system is expected to work with FOSS projects of various size in an automatic or semi-automatic manner. Particularly, the system should build and instrument FOSS

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 7: FOSS Projects Used in Scalability and Usability Experiment

projects with as little human involvement as possible.

Experiment Methods

We experimented with four FOSS projects of various scales (see Table 7) to evaluate the scalability and usability of our integrated system. For all the projects, we enabled code instrumentation of the aforementioned memory management security property.

Experiment Results

The following paragraphs document our experience of creating, instrumenting and executing these projects in the system.

- Amaya version 9.55:

Project creation was performed strictly following the description in Section 5.1.3.

Building and instrumenting the project required several prerequisites to be satisfied.

Firstly, a distribution specific directory (in our experiment *Linux*) needed to be cre-

ated. Secondly, the *Mesa* library distributed with the source code needed to be built

and installed in the system. Thirdly, a project Makefile needed to be generated by

autoconf utility through the "`./configure CFLAGS=-g`" command-line option.

Executing the project did not require extra effort. We followed the process described

in Section 5.1.3 to execute the program.

- gdLibrary version 2.0.35:

Creating the project required no additional effort to the process described in Section 5.1.3. A project Makefile needed to be generated by autoconf utility before building and instrumenting the project. As this is only a library, we could not execute it without extra effort.

- gzip version 1.24:

Creating and executing the project were performed following the normal process described in Section 5.1.3. Building and instrumenting the project also necessitated a project Makefile to be generated by autoconf utility.

- inetutils version 1.5:

The project creation, instrumentation and execution process of inetutils were identical to that of gzip.

The following points sum up our experiment on the scalability and usability of the integrated system.

- In general, automatic project building and instrumentation was achieved. As mentioned before, our current system anticipates an existing Makefile to drive project building. Hence, invocation of auto-configuration from command-line is accepted. We did not modify projects' Makefiles to enable automatic building. Three of the four projects were successfully built and instrumented based on the existing Makefiles.
- The exceptional case, i.e. Amaya, represents the scenario where a FOSS project uses

external libraries that are not available in the operating system. So far, we have not come up with any solution to this issue, because library dependency is project-specific.

- Experiment with gdLibrary demonstrates the limitation of our build-and-run approach. The current system can only work with executable FOSS projects. Extra efforts are needed in order to dynamically analyze the behavior of a library. For example, a driver program is to be constructed to use its APIs.

5.3 Summary

In this chapter, we introduced our integrated system for security property specification and dynamically checking properties on FOSS projects. We conducted two major experiments. Experiment 1 demonstrated the capability of our system to dynamically detect security vulnerabilities. Experiment 2 showed that our system can build, instrument and execute large-scale FOSS projects. With future extension of an automatic test suite generator, we expect our system to automatically check security properties of large-scale FOSS projects.

Chapter 6

Conclusion

In this thesis, we reviewed different approaches to detecting security vulnerabilities in software source code. We investigated several tools for statically and/or dynamically analyzing source code for security evaluation. We discussed the advantages and disadvantages of both approaches, and particularly presented the motivation of introducing an extensible dynamic security analysis tool to work in synergy with static tools. We discovered that few dynamic program analysis tools support detection of user-specified security properties.

Our research efforts resulted in an integrated software system that can check system-specific security properties. Particularly, we introduced a new mathematical model called *Team Edit Automata* with prototyped implementation for security property specification. An extended version of GCC is developed to assist in automatically instrumenting program monitors in testee programs. The current implementation allows security analysts to instrument arbitrary code at various security-sensitive program points. The integrated system supports management of multiple FOSS projects, automatic project instrumentation and execution, and error report and review.

Our integrated system is the first part of a long-term ongoing research project - TFOSS project. The expected benefit is to automatically detect security vulnerabilities in FOSS projects with synergic contribution from static security analysis tools. So far, our system has built up an infrastructure for dynamic security analysis. However, to achieve this goal, the following issues need to be addressed:

1. How can we take advantage of static analysis to reduce the runtime overhead of our instrumentation code?
2. How do we automatically generate test cases to drive the execution of instrumented program?
3. How can our tool collaborate with static security analysis tools to improve the accuracy and soundness of the analysis?

Further to these questions, we need to improve our system with more features, including more powerful instrumentation capability, automatic test suite generation, interface with static security analysis tools, etc.

Bibliography

- [1] autoconf. <http://www.gnu.org/software/autoconf/> (Date of Access August 23, 2007).
- [2] Dmalloc. <http://dmalloc.com/> (Date of Access: July 27, 2007).
- [3] Doxygen. <http://www.doxygen.org> (Date of Access: August 20, 2007).
- [4] Forrester research. <http://www.forrester.com/rb/research> (Date of Access: August 14, 2007).
- [5] GCC Core 4.2.0. <http://gcc-ca.internet.bs/releases/gcc-4.2.0/> (Date of Access: August 20, 2007).
- [6] GCC Front Ends. <http://gcc.gnu.org/frontends.html> (Date of Access: August 24, 2007).
- [7] Gnu-bash. <http://www.linuxjournal.com/article/2784> (Date of Access: July 10, 2007).
- [8] The GNU Compiler Collection. <http://gcc.gnu.org/> (Date of Access: August 24, 2007).
- [9] GNU Make. <http://www.gnu.org/software/make/> (Date of Access: July 23, 2007).
- [10] Insure++. <http://www.parasoft.com/> (Date of Access: September 3, 2007).

- [11] Klocwork. <http://www.klocwork.com> (Date of Access: September 2, 2007).
- [12] National vulnerability database. <http://nvd.nist.gov/> (Date of Access: August 2, 2007).
- [13] Puma online user's manual. <http://ivs.cs.uni-magdeburg.de/puma/UsersManual/HTML/node1.html> (Date of Access: August 12, 2007).
- [14] Splint. <http://www.splint.org/> (Date of Access: July 21, 2007).
- [15] The strategy design pattern. <http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/StrategyPattern.htm> (Date Of Access: August 23, 2007).
- [16] *IEEE Standard Glossary of Software Engineering Terminology*, volume IEEE Standard 610.12-1990. IEEE, 1990.
- [17] Alfred V. Aho and Jeffrey D. Ullman jt.author. *Principles of compiler design*. Addison-Wesley Pub., Reading, Mass. ; Don Mills, Ont., 1977.
- [18] O. H. Alhazmi, S. W Woo, and Y. K. Malaiya. Security vulnerability categories in major software systems. *Proceedings of the Third IASTED International Conference Proceedings Communication*, 2006.
- [19] Cyrille Artho, Viktor Schuppan, Armin Biere, Pascal Eugster, Marcel Baur, and Boris Zweimüller. Jnuke: Efficient dynamic analysis for java.

- [20] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *2002 Symposium on Security and Privacy*, pages 143–159, Berkeley, CA, May 12-15 2002 2002. Computer Systems Laboratory, Stanford University, Stanford, CA 94305, United States.
- [21] L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfering program monitors. *Software Security - Theories and Systems*, 2609:154–171, 2003.
- [22] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314. ACM Press, 2005.
- [23] Maurice H. Ter Beek, Clarence A. Ellis, Jetty Kleijn, and Grzegorz Rozenberg. Synchronizations in team automata for groupware systems. *Comput.Supported Coop.Work*, 12(1):21–69, 2003.
- [24] Nadia Belblidia, Mourad Debbabi, Aiman Hanna, and Zhenrong Yang. Aop extension for security testing of programs. pages 647–650, 2006. Canadian Conference on Electrical and Computer Engineering 2006.
- [25] Rudolf Berrendorf and Bernd Mohr. Pcl - the performance counter library: A common interface to access hardware performance counters on microprocessors. Technical report, Research Centre Juelich, 2003.
- [26] Luboš Brim, Ivana Černá, Pavlína Vařeková, and Barbora Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. volume 31, page 4, New York, NY, USA, 2006. ACM Press.

- [27] M. Debbabi C. Talhi, N. Tawbi. Execution monitoring enforcement for limited-memory systems. In *the International Conference on Privacy, Security and Trust*, 2006.
- [28] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press.
- [29] Sean Callanan, Radu Grosu, Xiaowan Huang, Scott A. Smolka, and Erez Zadok. Compiler-assisted software verification using plug-ins. In *2006 NSF Next Generation Software Workshop, in conjunction with the 2006 International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.
- [30] Benjamin Chelf, Dawson Engler, and Seth Hallem. How to write system-specific, static checkers in MeTaL. *SIGSOFT Softw.Eng.Notes*, 28(1):51–60, 2003.
- [31] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of c code, 2004.
- [32] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, New York, NY, USA, 2002. ACM Press.
- [33] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with*

- 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM Press.
- [34] Alan DeKok. Announce: Pscan, a simple security scanner. <http://seclists.org/bugtraq/2000/jul/0095.html>, July 2000.
- [35] Nachum Dershowitz and Zohar Manna. *Verification : theory and practice : essays delivered to Zohar Manna on the occasion of his 64th birthday*. Springer-Verlag, Berlin ; New York, 2003.
- [36] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-based access control for mobile code. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 38–48, New York, NY, USA, 1998. ACM Press.
- [37] Clarence Ellis. Team automata for groupware systems. In *GROUP '97: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 415–424, New York, NY, USA, 1997. ACM Press.
- [38] U. Erlingsson and F.B. Schneider. Sasi enforcement of security policies: a retrospective. *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, 2:287–295 vol.2, 2000.
- [39] M. Ernst. Static and dynamic analysis: synergy and duality. ICSE Workshop on Dynamic Analysis (WODA), Portland, Oregon, USA, May 2003.
- [40] D. Evans and A. Twyman. Flexible policy-directed code safety. *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 32–45, 1999.

- [41] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002.
- [42] Free Software Foundation, <http://gcc.gnu.org/onlinedocs/gccint/>. *The GCC Internals*, 1.2 edition, 2007.
- [43] Lars Marius Garshol. Bnf and ebnf: What are they and how do they work? <http://www.garshol.priv.no/download/text/bnf.html>.
- [44] A.K. Ghosh, T. O’Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 104–114, 3-6 May 1998.
- [45] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.
- [46] Michael Goulde. Open source becoming mission-critical in north america and europe. <http://www.forrester.com/Research/Document/Excerpt/0,7211,38866,00.html>, September 2006.
- [47] The SUIF Group. Suif compiler system. <http://suif.stanford.edu/> (Date of Access: August 30, 2007).
- [48] R. Hadjidj. *Analyse et validation formelle des systemes temps reel*. PhD thesis, Ecole Polytechnique, Montreal, Canada, 2006.

- [49] William G. J. Halfond and Alessandro Orso. Combining static analysis and runtime monitoring to counter sql-injection attacks. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM Press.
- [50] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, Boston, 3rd edition, 2007.
- [51] Clinton Jeffery, Wenyi Zhou, Kevin Templer, and Michael Brazell. A lightweight architecture for program execution monitoring. In *PASTE '98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 67–74, New York, NY, USA, 1998. ACM Press.
- [52] E. Eugene Schultz Jr., David S. Brown, and Thomas A. Longstaff. Responding to computer security incidents. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, July 1990.
- [53] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational analysis of run-time monitoring: Fundamentals of javamac. *Electronic Notes in Theoretical Computer Science*, 70(4):1–15, 2002/12.
- [54] Moonjoo Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Formally specified monitoring of temporal properties*, pages 114–122, 1999.

- [55] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [56] Klocwork. Klockwork wins infoworld 2007 technology of the year award. http://www.klocwork.com/company/releases/01_09_07.asp. (Date of Access: September 2, 2007).
- [57] Dieter Kranzlmuller. Event graph analysis for debugging massively parallel programs. <http://www.gup.uni-linz.ac.at/dk/thesis/html/thesis.html>, 2000.
- [58] L. Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, SE-3(2):125–143, 1977.
- [59] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security*, V4(1):2–16, 2005. M3: 10.1007/s10207-004-0046-8.
- [60] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, 1996.
- [61] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 137–151, New York, NY, USA, 1987. ACM Press.
- [62] D. Mahrenholz, O. Spinczyk, and W. Schroder-Preikschat. Program instrumentation for debugging and monitoring with AspectC++. *Object-Oriented Real-Time*

Distributed Computing, 2002. (ISORC 2002). Proceedings. Fifth IEEE International Symposium on, pages 249–256, 2002.

- [63] B. Miller, D. Koski, C. Lee, V. Maganty, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical Report 1268, University of Wisconsin-Madison, May 1998.
- [64] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.
- [65] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Language and Systems*, 27(3):477–526, 2005.
- [66] State of California. California performance review 2004. <http://cpr.ca.gov/report/cprprt/issrec/stops/it/so10.htm>.
- [67] U.S. Department of Justice. Creator of melissa computer virus sentenced to 20 months in federal prison. <http://www.justice.gov/criminal/cybercrime/melissaSent.htm>, May 2002.
- [68] opensource.org. The open source definition, July 2006. <http://www.opensource.org/> (Date of Access: August 17, 2007).
- [69] Stephen Ostermiller. Syntax highlighting library. <http://ostermiller.org/syntax/>.
- [70] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, Mass., 2002.

- [71] G. Ho C. Deane P.J. Mucci, S. Browne. Perfapi - performance data standard and api, 1999. <http://icl.cs.utk.edu/projects/papi/> (Date of Access: August 20, 2007).
- [72] Charles W. Rapp. The State Machine Compiler. <http://smc.sourceforge.net/> (Date of Access: September 2, 2007).
- [73] M. Ronsse and K. De Bosschere. Jiti : Tracing memory references for data race detection. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, editors, *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, 19-22 September 1997, Bonn, Germany*, volume 12, pages 327–334, Amsterdam, 1998. Elsevier, North-Holland.
- [74] Er Ro Rs. Purify: Fast detection of memory leaks and access errors. <http://citeseer.ist.psu.edu/291378.html>.
- [75] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
- [76] Herbert Schildt. *C: the complete reference*. Osborne/McGraw-Hill, Berkeley, 2000.
- [77] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
- [78] Benjamin Schwarz, Hao Chen, David Wagner, Jeremy Lin, Wei Tu, Geoff Morrison, and Jacob West. Model checking an entire linux distribution for security violations. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

- [79] Secure Software. Rough auditing tool for security (rats).
<http://www.fortifysoftware.com/security-resources/rats.jsp>.
- [80] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004.
- [81] David Stout. "youth sentenced in government hacking case". The New York Times, September 2000.
- [82] Computer Emergency Response Team. <http://www.cert.org/stats/>. (Date of Access: August 14, 2007).
- [83] Jay-Evan J. Tevis. *Automatic detection of software security vulnerabilities in executable program files*. PhD thesis, Auburn, AL, USA, 2005. Director-John A. Hamilton, Jr.
- [84] Jay-Evan J. Tevis and John A. Hamilton. Methods for the prevention, detection and removal of software security vulnerabilities. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 197–202, New York, NY, USA, 2004. ACM Press.
- [85] SD Times. http://www.sdtimes.com/static/sdtimes100_07_04.html. (Date of Access: August 14, 2007).
- [86] Syrine Tlili and Mourad Debbabi. A novel type and alias analysis for c safety and security. (Unpublished).

- [87] John Viega and Gary McGraw. *Building secure software : how to avoid security problems the right way*. Addison-Wesley, Boston, 2002.
- [88] Mahesh Viswanathan. *Foundations for the run-time analysis of software systems*. PhD thesis, Philadelphia, PA, USA, 2000. Supervisor-Sampath Kannan and Supervisor-Insup Lee.
- [89] J. Voas. Fault injection for the masses. *Computer*, 30(12):129–130, 1997.
- [90] J. Voas, A. Ghosh, G. McGraw, F. Charron, and K. Miller. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *Computer Assurance, 1996. COMPASS '96, 'Systems Integrity. Software Safety. Process Security'*. *Proceedings of the Eleventh Annual Conference on*, pages 250–263, 1996.
- [91] David Walker. A type system for expressive security policies. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267, New York, NY, USA, 2000. ACM Press.
- [92] James A. Whittaker and Herbert H. Thompson. *How to break software security : effective techniques for security testing*. Pearson/Addison Wesley, Boston, 2004.
- [93] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. Meca: an extensible, expressive system and language for statically checking security properties. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 321–334. ACM Press, 2003.

- [94] Michael Young and Richard N. Taylor. Rethinking the taxonomy of fault detection techniques. In *ICSE '89: Proceedings of the 11th international conference on Software engineering*, pages 53–62, New York, NY, USA, 1989. ACM Press.

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 [43] 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')+;

```

Appendix B

Rough GIMPLE Grammar

This appendix presents the rough GIMPLE grammar [42].

The Grammar

```
function : FUNCTION_DECL
          DECL_SAVED_TREE -> compound-stmt

compound-stmt : STATEMENT_LIST
               members -> stmt

stmt          : block
               | if-stmt
               | switch-stmt
               | goto-stmt
               | return-stmt
               | resx-stmt
               | label-stmt
               | try-stmt
               | modify-stmt
               | call-stmt

block        : BIND_EXPR
              BIND_EXPR_VARS -> chain of DECLs
              BIND_EXPR_BLOCK -> BLOCK
              BIND_EXPR_BODY -> compound-stmt

if-stmt     : COND_EXPR
              op0 -> condition
              op1 -> compound-stmt
```

```

        op2 -> compound-stmt

switch-stmt : SWITCH_EXPR
            op0 -> val
            op1 -> NULL
            op2 -> TREE_VEC of CASE_LABEL_EXPRs
                The CASE_LABEL_EXPRs are sorted by CASE_LOW,
                and default is last.

goto-stmt   : GOTO_EXPR
            op0 -> LABEL_DECL | val

return-stmt : RETURN_EXPR
            op0 -> return-value

return-value : NULL
            | RESULT_DECL
            | MODIFY_EXPR
            op0 -> RESULT_DECL
            op1 -> lhs

resx-stmt   : RESX_EXPR

label-stmt  : LABEL_EXPR
            op0 -> LABEL_DECL

try-stmt    : TRY_CATCH_EXPR
            op0 -> compound-stmt
            op1 -> handler
            | TRY_FINALLY_EXPR
            op0 -> compound-stmt
            op1 -> compound-stmt

handler     : catch-seq
            | EH_FILTER_EXPR
            | compound-stmt

catch-seq   : STATEMENT_LIST
            members -> CATCH_EXPR

modify-stmt : MODIFY_EXPR
            op0 -> lhs
            op1 -> rhs

call-stmt   : CALL_EXPR
            op0 -> val | OBJ_TYPE_REF
            op1 -> call-arg-list

call-arg-list : TREE_LIST
            members -> lhs | CONST

addr-expr-arg : ID
            | compref

```

```

addressable : addr-expr-arg
             | indirectref

with-size-arg: addressable
             | call-stmt

indirectref  : INDIRECT_REF
             op0 -> val

lhs          : addressable
             | bitfieldref
             | WITH_SIZE_EXPR
             op0 -> with-size-arg
             op1 -> val

min-lval     : ID
             | indirectref

bitfieldref  : BIT_FIELD_REF
             op0 -> inner-compref
             op1 -> CONST
             op2 -> val

compref      : inner-compref
             | TARGET_MEM_REF
             op0 -> ID
             op1 -> val
             op2 -> val
             op3 -> CONST
             op4 -> CONST
             | REALPART_EXPR
             op0 -> inner-compref
             | IMAGPART_EXPR
             op0 -> inner-compref

inner-compref: min-lval
             | COMPONENT_REF
             op0 -> inner-compref
             op1 -> FIELD_DECL
             op2 -> val
             | ARRAY_REF
             op0 -> inner-compref
             op1 -> val
             op2 -> val
             op3 -> val
             | ARRAY_RANGE_REF
             op0 -> inner-compref
             op1 -> val
             op2 -> val
             op3 -> val
             | VIEW_CONVERT_EXPR
             op0 -> inner-compref

condition   : val

```

```

    | RELOP
      op0 -> val
      op1 -> val

val
  : ID
  | invariant ADDR_EXPR
    op0 -> addr-expr-arg
  | CONST

rhs
  : lhs
  | CONST
  | call-stmt
  | ADDR_EXPR
    op0 -> addr-expr-arg
  | UNOP
    op0 -> val
  | BINOP
    op0 -> val
    op1 -> val
  | RELOP
    op0 -> val
    op1 -> val
| COND_EXPR
  op0 -> condition
  op1 -> val
  op2 -> val

```


Appendix C

Source Codes Used in Experiment 1

This appendix lists the source codes used in Experiment 1 - Checking Memory Management Vulnerabilities.

C.1 Security Property Specification for Memory Management Vulnerabilities

C.1.1 MemorySM.sm: State Machine for States of Program Memory

```
%{
// a state machine models the state transition of dynamic memory chunk
%}

%start      MemoryChunck::START
%class     MemorySM
%header    MemorySM.h

%map      MemoryChunck
%%

START
{
    allocate
    ALLOCATED
```

```

    { OkeyTransition(); }

    Default
    Error
    { UnknownError(); }
}

ALLOCATED
{
    doWrite(addr: unsigned int)
    [ctxt.IsValidAddress(addr) == true]
    nil
    { SetInitializationStatus(addr, true); }

    doWrite(addr: unsigned int)
    [ctxt.IsValidAddress(addr) == false]
    Error
    { OutOfBoundaryError(addr); }

    doRead(addr: unsigned int)
    [ctxt.IsValidAddress(addr) == true
     && ctxt.GetInitializationStatus(addr) == true]
    nil
    {OkeyTransition(); }

    doRead(addr: unsigned int)
    [ctxt.IsValidAddress(addr) == false]
    Error
    { OutOfBoundaryError(addr); }

    doRead(addr: unsigned int)
    [ctxt.GetInitializationStatus(addr) == false]
    Error
    { ReadUninitializedError(addr); }

    doDeallocate(addr: unsigned int)
    [ctxt.IsStartAddress(addr) == true]
    DEALLOCATED
    { OkeyTransition(); }

    doDeallocate(addr: unsigned int)
    [ctxt.IsStartAddress(addr) == false]
    Error
    { FreeInvalidMemory(addr); }

    Default
    Error
    { UnknownError(); }
}

DEALLOCATED
{
    doDeallocate(addr: unsigned int)

```

```

    [ctxt.IsStartAddress(addr) == true]
    Error
    { DoubleFreeingError(addr); }

doDeallocate(addr: unsigned int)
[ctxt.IsStartAddress(addr) == false]
Error
{ FreeInvalidMemory(addr);

Default
Error
{ UnknownError(); }
}

Error
{
    Default
    Error
    { UnknownError(); }
}

%%

```

C.2 Vulnerable C Programs Evaluated in Experiment 1

C.2.1 Program 1

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    const int BUFSIZE = 10;
    char staticBuf[BUFSIZE];

    // decrementing array index "i"
    // within the loop result in
    // out-of-boundary array access
    for (i = 0; i < BUFSIZE; ++i)
    {
        i -= 2;
        staticBuf[i] = i;
    }

    return 0;
}

```

C.2.2 Program 2

```
#include <stdio.h>

struct node
{
    struct node* next_;
};

void InitNode(struct node* n)
{
    n->next_ = 0;
};

void Cleanup(struct node** n)
{
    if ( (*n)->next_ )
    {
        free (&(*n)->next_);
    }

    free(*n);
};

int main(int argc, char *argv[])
{
    struct node* n1 =
        (struct node*)malloc(sizeof(struct node));
    struct node* n2 =
        (struct node*)malloc(sizeof(struct node));

    InitNode(n1);
    InitNode(n2);

    n2->next_ = n1;
    n1->next_ = n2;

    // circular reference between n1 and n2
    // upon Cleanup n2, double-freeing occurs
    Cleanup(&n2);

    return 0;
}
```

C.2.3 Program 3

```
#include <stdio.h>

void wrapped_read(char* buf, int count)
{
    fgets(buf, count, stdin);
}
```

```

void TaintedAccess(unsigned int dst_len)
{
    char buf1[12];
    char buf2[12];

    char* dst = (char*)malloc(dst_len);

    wrapped_read(buf1, sizeof(buf1));
    wrapped_read(buf2, sizeof(buf1));

    // possible buffer overflow
    // depending on user input from "len"
    sprintf(dst, "%s-%s\n", buf1, buf2);

    free(dst);
}

int main(int argc, char *argv[])
{
    unsigned int len;

    scanf("%d", &len);

    TaintedAccess(len);

    return 0;
}

```

C.2.4 Program 4

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned int bufsize = 0;

    printf("please specify buffer size: ");
    scanf("%d", &bufsize);

    const char* buf1 = "Hello world!";

    char* buf2 = (char*)malloc(bufsize);

    if (buf2)
    {
        // strcpy without checking destination buffer length
        // in this case buf2 is dynamically determined
        // buffer overflow may occur if user input of bufsize
        // is too small
        strcpy(buf2, buf1);
    }
}

```

```

    free(buf2);

    return 0;
}

```

C.2.5 Program 5

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned int bufsize = 0;

    printf("please specify buffer size: ");
    scanf("%d", &bufsize);

    const char* buf1 = "Hello world!";

    char* buf2 = (char*)malloc(bufsize);

    bufsize = bufsize * 2 + 1;

    // free statement in this if block never
    // executes due its previous line
    // this program is intentionally written to
    // be FREE of double-freeing vulnerability
    if (bufsize % 2 == 0)
    {
        free(buf2);
    }

    free(buf2);

    return 0;
}

```

C.2.6 Program 6

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    /*
    buffer[0] = 'h';
    buffer[1] = 'e';
    buffer[2] = 'l';
    buffer[3] = 'l';
    buffer[4] = 'o';
    buffer[5] = '\0';
    */
}

```

```
char buffer[] = "hello";  
buffer[5] = 'A';  
  
// depending on the platform and compiler  
// the following statement may print  
// an overflowed buffer  
printf("buffer string is: %s\n", buffer);  
  
return 0;  
}
```