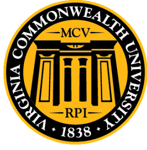

Virginia Commonwealth University
VCU Scholars Compass

Theses and Dissertations

Graduate School

2011

THE PROGRAM PATHING TRUST MODEL FOR CRITICAL SYSTEM PROCESS AUTHORIZATION

Robert Dahlberg
Virginia Commonwealth University

Follow this and additional works at: https://scholarscompass.vcu.edu/etd

Part of the Engineering Commons

© The Author

Downloaded from
https://scholarscompass.vcu.edu/etd/237

This Dissertation is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.

www.manaraa.com

Program Pathing Trust Model for Critical System Process Authorization

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering at Virginia Commonwealth University.

by

Robert Andrew Dahlberg
Master of Science, Computer Science, Northern Illinois University, 1982
Master of Arts, Philosophy, Northern Illinois University, 1982
Bachelor of Arts, Philosophy, Western Illinois University, 1976

Director: David Primeaux, PhD
Associate Professor
Computer Science Department

School of Engineering
Virginia Commonwealth University
Richmond, Virginia
April, 2011

المنارة للاستشارات
www.manaraa.com

Acknowledgement

I would like to thank my past teachers and employers for giving me opportunities, guidance and support that lead to this dissertation. And to thank all those colleagues, with whom I have collaborated with over the last 30 years, especially those with whom I first worked on the Program Pathing problem. Special thanks to Barry Schager and Eb Klemens (the S and K in SKK, Inc) for sharing their knowledge and granting many opportunities throughout my career.

Thanks to Dr. Robert Moore who inspired me while an undergraduate at Western Illinois University to pursue a graduate degree. Dr. Mason Myers (deceased) who while as his teaching assistant in logic encouraged me to study computer science. And thanks to Dr. Rodney Angotti, who gave me my first teaching position at the university level and has encouraged me many times throughout the years to pursue a PhD.

Special thanks to Lisa Porter for proof reading and editing this paper, to ensure its readability.

Thanks to my parents for their encouragement. They never let me think that it was an option not to go to college. I owe an especial debt to my grandparents, who took great pride in my academic achievement which in itself was a source of encouragement. And thanks to all my uncles who were engineers, many of who had never earned a degree but became respected engineers in their industries and created a legacy for me to follow.

I'd like to recognize my wife, Susan Dubuque, for all her moral support and constant reminding that pursuing a PhD was fun and that the journey was more important than the goal.

Thank my daughters, Maribeth, Caroline, and Rebecca for their love and support. It is to them that I dedicate this work.

And finally, I'd like to thank the faculty of VCU's computer science department for opening up the world of academic research. Special thanks to Dr. David Primeaux, my dissertation director, who's mentoring has made this dissertation possible. Dr. Primeaux has enriched the experience at VCU beyond just the pursuit of a PhD. His work on the creation of a master's degree in information system security and the creation of a security lab has made this experience at all the more valuable.

المنارة للاستشارات
www.manaraa.com

Table of Contents

TABLE OF CONTENTS II

LIST OF FIGURES VIII

ABSTRACT X

CHAPTER 1: INTRODUCTION 1

1.1 OVERVIEW 1

1.2 CONTRIBUTIONS 2

CHAPTER 2: BACKGROUND TERMINOLOGY AND DISTINCTIONS 4

2.1 PROCESSES 4

2.1.1 External Processes 4

2.1.2 Internal Processes 5

2.1.2.1 Operating System Processes 5

2.1.2.1.1 OS Kernel 5

2.1.2.1.2 OS Utilities 6

2.1.2.2 Application Processes 6

2.1.2.2.1 System Application Processes 6

2.1.2.2.2 User Application Processes 7

2.1.3 Process Behavior 7

2.1.3.1 Normal Process Behavior 8

2.1.3.2 Abnormal Process Behavior 9

2.2 PROCESS INVOCATION SEQUENCES 9

2.2.1 Valid Process Invocation Sequences 10

2.2.2 Invalid Process Invocation Sequences and System Integrity 10

CHAPTER 3: PROBLEM: WHY SYSTEM INTEGRITY IS IMPORTANT 12

المنارة للاستشارات

www.manaraa.com

3.1 Malware ... 12

3.2 Operator Error ... 14

3.3 Research ... 16

3.3.1 Requirements ... 16

3.3.1.1 Authentication ... 17

3.3.1.2 Authorization ... 17

3.3.1.2.1 Invocation ... 18

3.3.1.2.2 Static Process Invocation ... 18

3.3.1.2.3 Dynamic Process Invocation ... 19

3.3.1.3 Accountability ... 19

CHAPTER 4: SECURITY BACKGROUND ... 21

4.1 The Program Pathing Problem ... 21

4.2 Security and Program Pathing ... 24

4.2.1 Detection and Protection Systems ... 24

4.2.2 Aspects of Access Control Systems (Protection System) ... 25

4.2.2.1 Data Security ... 26

4.2.2.2 System Security ... 27

CHAPTER 5: PROGRAM PATHING BACKGROUND ... 28

5.1 Trusted System ... 28

5.1.1 What is a Trusted System? ... 28

5.2 Program Pathing as part of a Trusted System ... 29

5.2.1 Conceptual Security Models Related to Program Pathing ... 29

5.2.1.1 Goguen-Meseguer Model ... 29

5.2.1.2 Clark-Wilson Integrity Model ... 30

5.2.1.3 Brewer-Nash Model (Chinese Wall) ... 32

5.2.2 Other Implementations of a Trusted System Using Invocation Sequences ... 33

5.2.2.1 ACF2® ... 37

www.manaraa.com

5.2.2.2 RACF® PADS ... 39

5.2.2.3 Top Secret® ... 40

5.2.3 More Recent Background ... 41

5.2.3.1 Trusted Path Execution (TPE) ... 41

5.2.3.2 Symantec's Critical Program System (CPS) ... 42

5.2.3.3 SELINUX ... 43

5.2.4 Current Literature on Program Pathing ... 47

5.2.4.1 Non-Computational Theory Approaches ... 48

5.2.4.1.1 Hofmeyr-Forrest – N-Gram Approach ... 48

5.2.4.1.2 Warrender - Forrest – Alternate Data Models ... 50

5.2.4.1.3 Ghosh – ANN Approach ... 51

5.2.4.1.4 Ammons -Larus – Retrieval Tree Approach ... 52

5.2.4.2 Computational Theory Approach ... 54

5.2.4.2.1 Ko-Fink – Execution Monitoring ... 54

5.2.4.2.2 Kosoresow-Hofmeyr – System Call Traces ... 56

5.2.4.2.3 Sekar – Finite State Automata Approach ... 58

5.2.4.3 Context Free Grammar or Pushdown Automata ... 60

5.2.4.3.1 Feng - Kolesnikov –Pushdown Automata Approach ... 61

5.2.4.3.2 Wagner- Dean – Pushdown Automata Approach ... 62

CHAPTER 6: PPT THEORETICAL MODEL ... 65

6.1 CRITERIA FOR A COMPUTATIONAL MODEL ... 65

6.1.1 . Necessary & Sufficient ... 66

6.1.2 Choosing a Computational Model ... 67

6.1.2.1 Multitasking Requirement ... 68

6.1.2.2 Regular Language ... 69

6.2 APPROPRIATE REPRESENTATION OF THE PROBLEM ... 72

6.2.1 Finite State Automata Representation ... 72

6.2.1.1 States Q ... 73

المنارة للاستشارات
www.manaraa.com

6.2.1.2 Alphabet Σ and process invocations 74

6.2.1.3 Transition Relation Δ 77

6.2.1.4 Start State P_0 80

6.2.1.5 FSA Issues 80

6.2.2 Regular vs. Context Free Language 82

6.3 FINITE STATE AUTOMATA AND THE PROGRAM PATH TRUST MODEL 82

6.3.1 Why Finite State Automata is a better Computational Model choice. 84

6.3.2 Finite State Automaton PPT Representation 84

6.3.3 PPT Finite-state Automata Learning Mode 86

6.4 RELATION BETWEEN L_V AND $L(DFA_T)$ 90

CHAPTER 7: IMPLEMENTATION OF THE PPT MODEL 92

7.1 ALTERNATIVES FOR IMPLEMENTING THE PROGRAM PATHING TRUST DFA 93

7.1.1 PPT DFA Bit Map Implementation 93

7.1.2 PPT DFA Adjacency-List Implementation 96

7.2 MEASURING IMPLEMENTATION STRUCTURES 97

7.3 CODING STRUCTURES IN PPTM 99

7.3.1 PPTM Basic Structure 99

7.3.2 How the PPTM works 101

CHAPTER 8: DEVELOPMENT AND TEST RESULTS 104

8.1 DEVELOPMENT 104

8.2 UNIT TESTING AND DEBUGGING 105

8.2.1 Test Reading Training Data and Building the Automata Structure 105

8.2.2 Verify Data Against the Profiled Training Data in the Automata 106

8.3 SYSTEM TESTING 106

8.3.1 Tstdata – Random Test Data Generator 107

8.3.2 Performance Testing the PPTM prototype 108

المنارة للاستشارات

www.manaraa.com

CHAPTER 9: FUTURE RESEARCH 111

9.1 Implement PPTM into the Operating System's Kernel 111

9.2 Testing 111

9.3 Process Authentication 112

9.4 The Validity of Inferred Process Invocation Sequences 112

CHAPTER 10: CONCLUSION 113

10.1 Computational Theory Approach to Validating Process Invocation Sequences 113

10.1.1 Required Computational Power 114

10.1.2 Translating Theory into Solutions 114

10.2 Impact upon the Program Pathing Problem 115

10.2.1 Mapping Process Authority to Invoke Processes 115

10.2.2 Mode Characteristics of Some Process Invocations 116

10.3 Potential Use of the Program Pathing Trust Model 117

10.3.1 Program Pathing in an Access Control System 117

10.3.2 Program Pathing in a System Integrity System 117

REFERENCES 119

APPENDIX A: PROTOTYPE SOURCE CODE 127

A.1 PPTM Source Code 127

A.2 PPTM Automata Header files 137

A.3 PPTM Print Header file 138

A.4 Testdata (automated data creation) Source Code 140

A.5 Test Header file 142

APPENDIX B: GLOSSARY 143

APPENDIX C: ACF2® PROGRAM PATHING DEFINTION MODULE 144

المنارة للاستشارات

www.manaraa.com

APPENDIX D: PROCESS AUTHENTICATION ***150***

D.1 Ownership Authentication Factor *151*

D.2 Inheritance Authentication Factor *151*

D.3 Location-based Authentication Factor *152*

APPENDIX E: IS $Lv \cup LDFAt$ = { }? ***153***

10.4 Other Approaches Making Assumptions similar to Is $Lv \cup LDFAt$ = { } *153*

E.2 Impact of the Assumption *154*

VITA ***156***

المنارة للاستشارات

www.manaraa.com

List of Figures

Figure 3-1 New Malware Code Threats - Symantec 13

Figure 3-2: IDC's Survey of External vs. Internal Threats 14

Figure 5-1 MVS Control Block Structure that ACF2 Program Pathing Maps 38

Figure 5-2 SELinux Domain - Resource Concept 44

Figure 5-3 SELinux Policy Reference Language for daemon.te 45

Figure 5-4 Conceptual Diagram of Daemon Policy Reference Example 45

Figure 5-5 SELinux Process Control 46

Figure 5-6 SELinux Process Transitions 47

Figure 5-7 Literature Mapping 48

Figure 5-8 Seka's FSAFigure 59

Figure 6-1 Chomsky's Hierarchy of Formal Languages 65

Figure 6-2: Interleaved Process Execution 69

Figure 6-3: Definition of Regular Expression 71

Figure 6-4: Representation of the FSA Recognizing the Union of Languages S1 and S2 72

Figure 6-5: Definition of Deterministic Finite State Automata 73

Figure 6-6 Initial Start State – NFA_{t0} 77

Figure 6-7 NFA_{t1} 78

Figure 6-8 NFA_{t2} 78

Figure 6-9 Building the Valid Process Invocation language 79

Figure 6-10 $DFA_{t=0}$ Transition Diagram 85

Figure 6-11: Initial PPT DFA Translation Table 86

Figure 6-12 Transition Diagram Representing the DFA Recognizing Language $DFA_{t=3}$ 87

المنارة للاستشارات

www.manaraa.com

Figure 6-13 $DFA_{t=3}$ Transition Table Learning 88

Figure 6-14 $DFA_{t=5}$ Transition Diagram 88

Figure 6-15 $DFA_{t=5}$ Transition Table 89

Figure 6-16: Algorithm for Building PPT DFA 90

Figure 7-1: Algorithm for PPT DFA in Validation Mode 92

Figure 7-2 Program Pathing Bit Map 93

Figure 7-3: Adjacency-matrix 94

Figure 7-4: Adjacency-list 96

Figure 7-5: Memory and Process Representation Comparisons 97

Figure 7-6: PPTM Anchor Data Area 100

Figure 7-7: Automata Data Structures 101

Figure 7-8: “automata” Data Area Containing the Process Invocation Sequences 102

Figure 7-9: Format of the “*automtrace*” file 103

Figure 8-1 System Test Results 109

Figure E-1: Process Invocation Example 155

المنارة للاستشارات

www.manaraa.com

ABSTRACT

THE PROGRAM PATHING TRUST MODEL FOR CRITICAL SYSTEM PROCESS AUTHORIZATION

By Robert A. Dahlberg, MS, MA

A dissertation proposal submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2011

Dissertation Director: David Primeaux, PhD. Associate Professor, Computer Science

Since computers are relied upon to run critical infrastructures – from nuclear power plants to electronic battlefield simulations – the concept of a "trusted" or tamperproof system has become even more important. Some applications have become so critical that it is imperative that they run as intended, without interference. The consequences of these systems not running as intended could be catastrophic. This research offers a solution for a key element for protecting these critical servers – validating process invocation sequences.

The purpose of this research is to increase operating system security by detecting, validating, and enforcing process invocation sequences within a critical system. If the processes on a critical system are not those that are intended to run or support the critical system, or if a system is able to run processes in an unauthorized sequence, then the system is compromised and cannot be trusted. This research uses a computational theory approach to create a framework for a solution for the process invocation sequence problem. Using the Program Pathing Trust Model, a solution capable of identifying both valid and invalid process invocation sequences is developed.

www.manaraa.com

Chapter 1: Introduction

Computer security emerged as an area of interest around 1967 [SCC70] [CST72] [Schr74-1] [Schr74-2]. As computers became increasingly utilized in government and private industry, they became indispensable. The need for computer security has become more evident with the increased prevalence of malware combined with societal dependence upon computers. As computers are relied upon more to run critical infrastructures – from nuclear power plants to the electronic battlefield – the concept of "trusted" or tamperproof systems has become even more important. Critical applications must run as intended, without interference, or the consequences could be catastrophic. Power grids could go offline, transportation systems could fail, battlefield controls could black out or the nation's financial transactions could stall, resulting in scenarios such as loss of human life and financial losses.

1.1 Overview

Early investigators discovered that to be effective security systems must work in a symbiotic relationship with the operating system (OS). The OS relies upon the security system to ensure that OS integrity is maintained. And the security system relies upon OS integrity not to let other facilities interfere with or circumvent it.

The purpose of this research is to develop a security solution model for maintaining system integrity, meaning that system integrity is maintained by permitting a system to execute only *normal processes* in *valid process invocation sequences*. The terms *normal process* and *valid process invocation sequence* are explained in detail later in section 2.2. OS security is increased by ensuring that only trusted process invocation sequences are executed within the system. If the

المنارة للاستشارات
www.manaraa.com

processes running on the system are not those intended or if the system is able to run processes in an unacceptable sequence, then the system is compromised and cannot be trusted.

1.2 Contributions

This research takes a unique approach to the problem of insuring the integrity of a critical system. While other approaches (described in chapter 5) focus upon determining whether a previously encountered sequence of processes is valid, the approach in this research validates each process's authority to invoke a subsequent process, thereby adding a new dimension to access control. Prior approaches to access control do not address the validation of a process's authority to invoke other processes.

The program pathing trust (PPT) model developed provides a theoretically sound framework for assessing the validity of process invocation sequences. While other research has employed theory-based structures such as automata without explicit discussion of the required computational power, this research develops a theory-based approach with respect to the security issue of validating process invocation sequences. This research shows that the computational power of a Finite State Automaton is sufficient because process invocation sequences have the structure of a Regular Language.

The PPT model resulting from this research is more compact than several previously suggested models. In the PPT model each process is represented only once. This is not the case in other approaches. Furthermore, where other approaches provide only a method determining whether a previously encountered sequence of processes is valid, the PPT model can be used not only to similarly assess whether some candidate process invocation sequence is valid, but also to reject a set of invalid process invocation sequences, whether or not previously encountered, and also to infer the possible validity of some process invocation sequences that have not been

المنارة للاستشارات
www.manaraa.com

previously encountered.

المنارة للاستشارات
www.manaraa.com

Chapter 2: Background Terminology and Distinctions

The terminology and distinctions made in this section are used to describe and define the problem and the proposed solution.

2.1 Processes

The basic function of a computer is to execute programs. A *program* is a set of machine instructions that are organized in a logical sequence to perform a task or process [Stall92]. A *process* is a program that is loaded into main memory and executed [Silb05]. The operation of a computer may be modeled as a series of processes invoking other processes [Stall92]. Other than physical threats to a computer, a process is required in order to pose a threat to a computer. Therefore, it is a fundamental premise of this research that all threats to a computer that are of interest are associated with *processes* and the *invocation sequences of processes*.

2.1.1 External Processes

External processes are processes that have not been explicitly installed by a system administrator. These processes might be applets loaded by users visiting a webpage, scripts or programs written (or downloaded) by users, macros in an application (like Mircosoft Office®), or malware that has otherwise infiltrated the system. *External processes* can pose a danger to a system because they may come from unknown sources. For this reason they are generally not desirable [CSI03] [Eete08].

External processes tend to be a security concern more for workstations than for critical servers. However, even critical servers can be susceptible to *external processes*. Poor access control can allow a user or a process to install an *external process* into a restricted directory. Or,

المنارة للاستشارات
www.manaraa.com

system vulnerability can be exploited to implant an *external process* into a directory or into an execution sequence. *External processes* pose an obvious threat.

2.1.2 Internal Processes

Internal processes are processes that have been intentionally installed on a computer system by a system administrator. *Internal processes* are often part of a vendor-supplied software package. They are usually purchased from and supported by a commercial vendor but may be open source software. On a critical server, ideally only software that is critical to the function of the system should be installed. There may be, however, processes included in the installed software that are not used as part of the critical function of the system. Many software packages have features that are not needed by a particular enterprise and are therefore not used. The processes that support these features may be installed on the computer, but may not be executed.

2.1.2.1 Operating System Processes

Operating system processes are *internal processes* that are responsible for the management of computer resources (hardware, memory, I/O and intercommunication), the coordination of system activities and the sharing of the computer resources. The operating system acts as a host for all other processes that run on the machine [Stall92]. The OS is composed of a number of processes (such as services), not all of which are needed by a critical system – although in a full installation they reside on the system.

2.1.2.1.1 OS Kernel

The *OS kernel* is a set of core OS processes. They perform the most critical functions of the OS, and without them no other processes could execute. The OS *kernel* is made up of those processes that manage the execution of other processes in the OS. They perform process,

المنارة للاستشارات
www.manaraa.com

memory and I/O management and other OS support functions such as interrupt handling, auditing and monitoring. Kernel processes with the highest authority execute in "system" (or "kernel") mode. System mode allows kernel processes to execute privileged instructions and be exempt from access controls [Stal92]. Any process that maliciously modifies a kernel process is referred to as a rootkit. Rootkits are designed to allow another process to gain elevated authority to circumvent the system's data and system security [Hogl05]. If an OS *kernel* process is compromised, the entire OS is generally un-useable and has to be reinstalled, unless the compromised process is identified, and removed (or replaced).

2.1.2.1.2 OS Utilities

OS utilities are also internal processes that are part of the operating system. These processes are usually invoked by terminal commands or through a user-initiated GUI. The processes are loaded from the installed operating system directories. These directories usually require elevated authority to update, and are therefore considered reliable. OS *utilities* may or may not run with elevated authority. These processes are also vulnerable to rootkits.

2.1.2.2 Application Processes

Application processes are internal processes that a system administrator has installed on the system and expects to run as an integral part of the system's primary function. These *application processes* may or may not run with elevated privileges.

2.1.2.2.1 System Application Processes

System applications (sometimes referred to as middleware) can be defined as application processes that are installed to support a user application. These applications are neither part of the OS nor the *user applications* (described below) that they support. *System applications* such

المنارة للاستشارات
www.manaraa.com

as a database or a data transport system add more sophisticated functions than the OS alone is designed to provide. These processes, like OS processes, usually require elevated access and the directories they reside in are restricted. However, as in the case with OS processes, there are generally unnecessary features, utilities, application program interfaces or sample code that could be used to interfere with the processes within the application's primary function. These extraneous processes pose a possible threat if run, as they can steal CPU cycles or otherwise interfere with the application [Bre89] [Gogu82].

2.1.2.2.2 User Application Processes

User applications are application processes that provide the reason why all the other processes exist. On a critical system, only necessary *user applications* should be installed. These processes may or may not need elevated access to execute. The directories in which they reside must be protected with appropriate access control techniques.

User applications can be vendor supplied or developed in-house. Vendor-supplied user applications can cause the same concerns as *system applications* and *OS processes* with respect to their including extraneous content. An in-house developed *user application*, however, is likely leaner in its deployment and only deploys those processes that are required by users of the system. Therefore, in-house developed applications would be less likely to contain unnecessary processes that might be executed and compromise the system. However, in-house user applications require good version control because poor version control can introduce vulnerability.

2.1.3 Process Behavior

Each of the preceding process types are classified as either having *normal* or *abnormal process behavior*. In this research, a *process's behavior* is defined as the execution of its

المنارة للاستشارات
www.manaraa.com

sequence of machine instructions. A process can manipulate memory, invoke OS services or invoke other processes. The process's logic may provide multiple execution paths, not all of which may be desirable in a particular environment. Desired behavior is that behavior that is designed into the process to fulfill the mission of the organization. Every process has a function that an organization intends it to accomplish. In this paper, a *normal process* is defined as a desired process running on a critical server.

2.1.3.1 Normal Process Behavior

The problem is broader in scope than previous related security research in intrusion detection, which focuses only on malware intrusion. This research focuses on the larger problem of system integrity. What would normally be a false positive for an intrusion detection system may prove not to be such in this research. The distinction lies in the definition of *normal behavior*. *Normal behavior* for an intrusion detection system generally means the execution of any software that is intentionally installed by authorized users. The purpose of such an intrusion detection system is to identify any other software that has infiltrated the system. For a critical server, however, that definition is insufficient. This research defines *normal behavior* as resulting from only those processes that are intended or are necessary to run on a system to achieve its intended function. Thus, a process that may be considered part of *normal behavior* in another system may not be considered normal in a critical system. For example, because only processes that are necessary for the fulfillment of a critical system's function should be allowed to run, it may not be acceptable for a critical system to allow SMTP (email) traffic processing. This reduces superfluous processes executing and taking up valuable system resources or otherwise interfering with critical functions.

المنارة للاستشارات
www.manaraa.com

2.1.3.2 Abnormal Process Behavior

Abnormal processes are defined as the complement of the set of *normal process*es. All processes are assumed to be *abnormal* unless they are determined necessary and appropriate to support the mission for which the server was built. For example, a critical system created to run a company's accounting system probably shouldn't be allowed to execute processes to run the company's emails. Even an *internal process* installed as part of the OS or an application can be considered *abnormal*, if it is not a process necessary to achieve the system's intended function. Thus it is not necessary that a process be *external* in order for it to be labeled *abnormal*.

Abnormal processes (internal or external) can also be new processes that infiltrate the system, or ones that masquerade as *normal* processes. New processes that infiltrate the system would most likely be *external processes*. They can be a validly loaded process such as an applet which might be an unknown process loaded into a JAVA virtual machine from across the network. When an *abnormal* process masquerades as a *normal* process, it is generally malware or possibly a variant of a *normal* process. A system does not maintain its integrity if it runs any *abnormal* processes.

2.2 Process Invocation Sequences

A computer system executes a sequence of processes. As part of a *normal process's* behavior, it might invoke one or more processes for OS system services or another application process; at some times, for some processes, this sequence is significant. The execution of some processes should not occur in an unconstrained order, but rather within a range of acceptable orders.

The OS provides a process scheduler that manages all *process invocation sequences*. From the time the OS is booted, the computer executes a process sequence. The various orderings of

المنارة للاستشارات
www.manaraa.com

processes that may be scheduled for execution by the OS scheduler represents a set of *process invocation sequences*. Ensuring that some process invocations execute in order is as important as ensuring that a process executes its machine instructions in the correct order. Determining whether a *process invocation sequence* is *valid* or *invalid* is the central theme of this research.

2.2.1 Valid Process Invocation Sequences

Valid process invocation sequences are defined as those process invocation sequences that invoke a set of *normal processes* in an order that accomplishes or supports the system's intended primary function. In defining *normal processes*, a server dedicated to running accounting functions should run only accounting processes and those processes necessary to support those accounting functions. However, in addition, every *normal process* supporting this accounting function should be coded to invoke only certain processes in a limited range of order. A process's logic may support different logic paths, but the number of processes it may invoke is finite (although, perhaps large), whether these processes include another application process or an *OS process*. As the critical server executes its primary function, only a subset of all possible *process invocation sequences* supports the intended functions of the system. This subset consists of exactly the *valid process invocation sequences*.

2.2.2 Invalid Process Invocation Sequences and System Integrity

System integrity can be compromised by *normal processes* running in an *invalid process invocation sequence*. The set of *invalid process invocation sequences* is the complement of the set of *valid process invocation sequences*. A system is said to maintain its system integrity if it runs only *normal processes* in *valid process invocation sequences*.

المنارة للاستشارات
www.manaraa.com

The classic example of a *valid* and *invalid invocation sequences* is illustrated in z/OS MVS[1]'s AMASPZAP. AMASPZAP is a program that allows a system program to modify machine instructions at the binary level. The system program can verify the binary instruction codes and change them using AMASPZAP. AMASPZAP, which modifies machine instructions, is a *normal* maintenance *process*. If that process is invoked from the SMP/E[2] process, the process invocation sequence is considered *valid*. The SMP/E process provides restricted access control that AMASPZAP alone does not provide. Therefore, any invocation of AMASPZAP that is not made directly from SMP/E is considered a system integrity breach. If the AMASPZAP process is invoked directly from a TSO/E[3] process or some other process, then the *process invocation sequence* must be considered *invalid*.

The normality of a process invocation process can also be dependent on the wall-clock time at which it is executed. For example, a *process invocation sequence* may be *valid* if it runs during a system maintenance window (say: Saturday evenings 10:00PM to 4:00AM), but execution at another time should be considered an *invalid process invocation sequence*.

[1] z/OS MVS is IBM's MVS operating system which runs on the z10 chip and is the most recent descendent of the System/390, System/370 and System/360 chip series. z/OS is the most current version of the MVS operating system that runs on the z10 chip. MVS is the standard operating system used on the IBM mainframe for the last 40 years. [Webb08]
[2] SMP/E is IBM's System Modification Program/Extended. SMP/E is a tool for installing and maintaining software and for managing the inventory of software that has been installed on a Z/OS machine. [IBM08]
[3] TSO/E is a z/OS Base Time Sharing Option/Extensions (*TSO/E*) element that provides an interactive terminal interface. Equivalent to Putty or terminal services in Unix. [IBM09]

المنارة للاستشارات
www.manaraa.com

Chapter 3: Problem: Why System Integrity is Important

When addressing the problem of maintaining system integrity, it is important to understand what might cause a system to become compromised. The OS does provide some system integrity internally that is effective as long as programmers and administrators create and execute only *internal processes* according to some basic security principles [Harr03]. Of course, the only reasonably sure protection from programmer mistakes is to require stringent reviews of their code and adequate quality assurance verification to ensure these principles have been followed. However, even if these basic security principles are followed, system integrity remains challenged by *external processes* and users. Exposure to *external processes* can cause a system to be infected with malware. System integrity can also be compromised by access from administrators with privileged access.

3.1 Malware

Nearly all computers have some exposure to the Internet and, as a result, are under constant threat of attack by viruses, parasites, worms, Trojan horses, adware, bots and other intentionally designed malware. Symantec, a prominent security company, has tracked and documented the number of malware incidents per year is growing (see figure 3-1 below) [Syma10]. Almost all computers encounter outages or suffer poor performance due to malware. Countless time and money has been spent fixing, reimaging or replacing systems that have been compromised. Even with a defense-in-depth strategy using anti-virus, anti-spyware, anti-malware, intrusion detection, vulnerability assessment and access control security tools, new and innovative malware still penetrates through the defenses. However most users accept the risk, and consider it part of the cost of doing business [Eete08]. Even application servers buried deep in an organization’s infrastructure (such servers which are of most interest to this research) have some

المنارة للاستشارات
www.manaraa.com

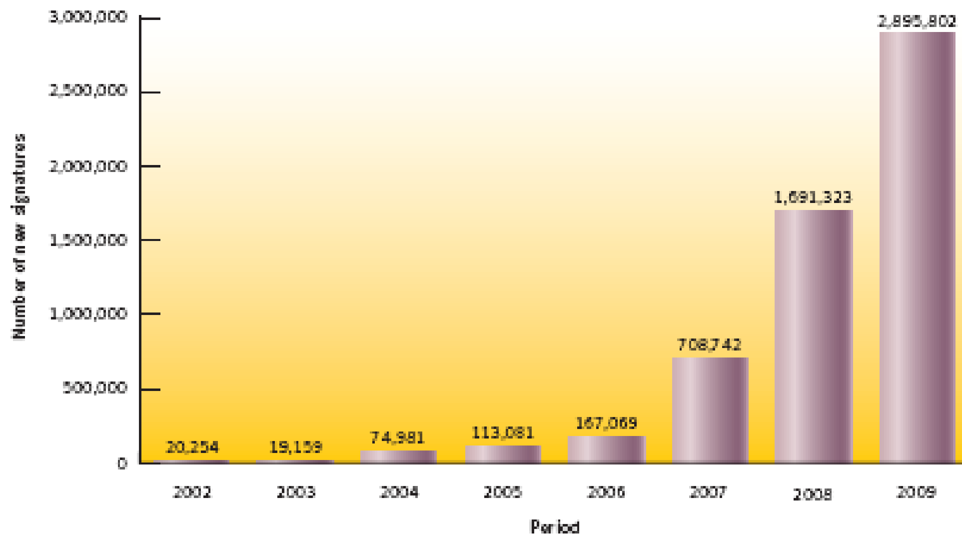

Figure 3-1 New Malware Code Threats - Symantec

exposure to the constant barrage of malware.

Some critical systems not only have to be concerned with the possibility of random malware attack, but also with a relatively high likelihood of attacks specifically targeted by cyber criminals or terrorists. Some systems support critical functions such as providing an electronic battlefield, balancing power grids, coordinating air traffic or regulating the money supply. Because of the critical applications they support, these systems cannot afford to be compromised, and therefore, warrant a stronger defense. Therefore, some defenses that normally would not be cost-effective on other systems are required on these systems. Fortunately, these critical systems are more likely to run on dedicated computer systems and can be more tightly controlled.

Malware infects systems by either implementing themselves as a new process within a *valid sequence of process invocations* or by masquerading as a known process within an apparently *valid process invocation sequence*. Ensuring that all processes and/or sequences of processes are *normal* becomes critical to verifying that malware has not infected a system. Preventing malware contamination is a by-product of ensuring system integrity.

المنارة للاستشارات
www.manaraa.com

3.2 Operator Error

Most prior work in OS security has been focused on intrusion detection [Amm98] [Appf04] [Feng03] [Forr96] [Gho00] [Hof98] [Ko94] [Kos97] [Wag01] [Warr99] and has been a reaction to the emergence of malware. Malware, although an important aspect of system integrity, is not the only concern. Operator error or internal threats can also compromise system integrity.

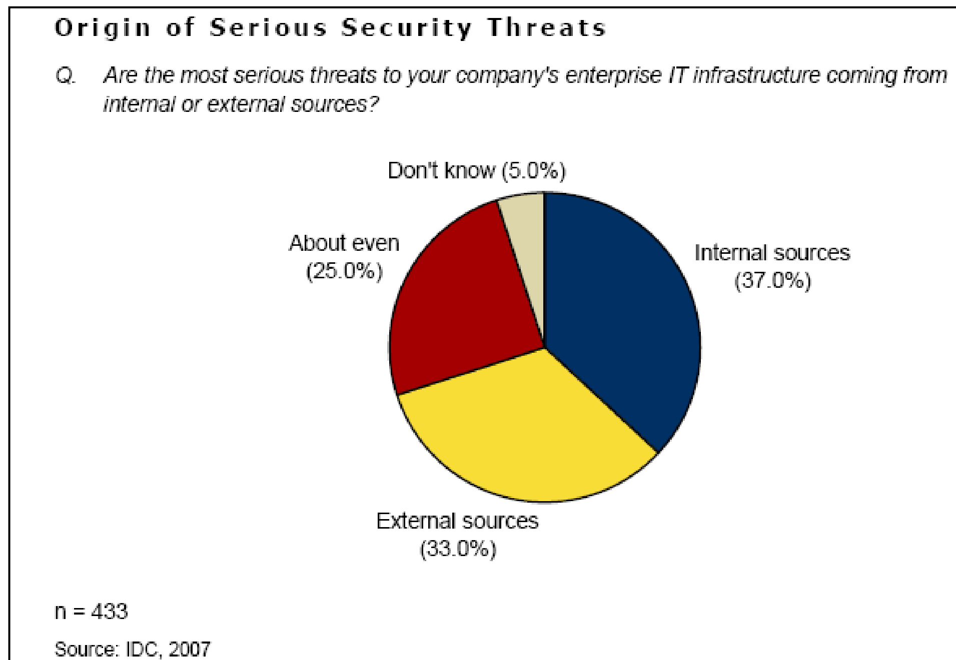

Figure 3-2: IDC's Survey of External vs. Internal Threats

Computer operators and security practitioners know that external threats are not the only threats to critical computer systems. In fact malware is not the main reason critical production systems fail or encounter production outage incidents. Production outage incidents are known to be caused more frequently by operator error or to occur after system maintenance or other changes are introduced to a system [Chri08] [CSI03] [Keen05]. Users with elevated privileges can pose a more serious threat to a system than malware because of their access using administrative authority. Security professionals know that historically the most dangerous threats to computer systems are internal, particularly for critical systems located deep in the

المنارة للاستشارات
www.manaraa.com

infrastructure [CSI03]. In 2007, International Data Corporation (IDC) research found that most threat focus was on external threats. A 2008 IDC report [Chri08] (figure 3-2) showed a shift of focus from external threats to internal threats.

Since it is difficult to predict the variety of things a system administrator might need to do to a system, they are granted higher privileges to enable them to fix or tune a system, which also allows them to interfere with a system's intended function by mistake. Even the best technicians make mistakes, sometimes with catastrophic effect [Chri08]. With a lack of understanding of how exactly the system works, or by simply hitting the ENTER key by mistake, technicians can unintentionally interfere with a critical system. Or technicians could submit a task that, although otherwise benign, could consume valuable CPU cycles needed for intended execution of the critical application.

System changes that may impact a critical system are normally reserved for a maintenance window, a time when production processes are not executing. At these times, the critical system is more tolerant of executing non-production associated processes. Administrators, however, need a security mechanism that would allow them to maintain a system during maintenance windows or when a system needs their intervention, such as when a system exhibits problems and needs an emergency fix. The security mechanism must not allow the administrator to run any process that may deviate from the normal production process during the hours when production processing is running. Maintenance processes should only run at specific wall-clock times within specific process invocation sequences and not while production process and process sequences are running.

المنارة للاستشارات
www.manaraa.com

3.3 Research

Other research in this field has taken a purely engineering approach. Researchers define a solution by focusing upon intrusion detection [Feng03] [Forr96] [Gho00] [Hof98] [Ko98] [Kos97] [Sek01] [Wag01] [War99] and focus upon resolving the malware problem only. These approaches are quick to formulate a solution to the problem of indentifying malware in process invocation sequences. While existing research addresses an important aspect of the problem, there is far more to this complex problem. “The engineer's first problem in any design situation is to discover what the problem really is” [Beak69]. This research analyzes the problem from a different perspective and then defines requirements to solve the problem(s) by developing a solution model.

There is a need for a security model to enforce system integrity by adding to the defense-in-depth arsenal that protects against malware and provides safeguards against technician errors. The facility must adhere to some basic security principles: it must perform authentication and authorization, and provide accountability. This research defines a solution model for a facility to provide system integrity controls, not only to mitigate malware intrusions, but also to provide control over technicians so that they can only apply changes during maintenance windows or in emergency situations. This research analyzes the system integrity problem, analyzes which computational model is necessary and sufficient to address the problem, defines the requirements, creates a solution model, identifies the kinds of features needed in such a system, and tests a prototype of the solution.

3.3.1 Requirements

As part of analyzing of the problem of system integrity, the system requirements are identified. In this research, these requirements are based upon the AAA security principle:

المنارة للاستشارات
www.manaraa.com

Authentication, Authorization and Accountability. The AAA principle is an industry-accepted standard associated with all security solutions. Although the principle became best known through the AAA protocol RFCs defined by the IETF [Ietf00], it has been a guiding principle since the first access control systems were developed in the early 1970s. AAA is an accepted principle in the development of all security controls [Fire03].

3.3.1.1 Authentication

Authenticating the identity of processes is a prerequisite to verifying that a process is *normal* and that it belongs to a *valid process invocation sequence*. Unless a process is authenticated, another process can masquerade in place of a *normal process*. This is a requirement overlooked in other research concerned with mapping invocation sequences. Process authentication is defined in section 9.3 and a discussion of authentication methods are discussed in more detail in Appendix D. Although authentication of processes is very important, this component of security is not the focus of this research. This research assumes all processes presented to the scheduler are correctly authenticated.

3.3.1.2 Authorization

Process authorization (validating processes and invocation sequences) is a critical element of this research. An authenticated process is evaluated as to whether it is authorized to be invoked by the process that invoked it. Each process corresponding to the entire prefix (the portion of the sequence preceding the process to be authorized) of the process invocation sequence must be authorized in order for the executing process to be authorized. If the scheduler determines that a process is not invoked by a process that is authorized to invoke it, then the process is not scheduled for execution. If the process is *abnormal*, or any part of the prior *process invocation sequence* is *invalid*, then the invoked process is determined to be unauthorized. This research

المنارة للاستشارات
www.manaraa.com

focuses primarily on developing a model for determining the authorization of *valid process invocations*. Process authorization has two distinct functions: (1) to learn or define a *valid process invocation sequence* and (2) to verify that a current running process is the product of a *valid process invocation sequence*.

Determining a *valid invocation sequence* has been one of the most challenging aspects of this research. Other projects have used a variety of methods (see chapter 5), resulting in mixed success. One of the problems with determining the validity of a process invocation sequence is that today's systems are so complex that it seems no individual really knows what a *valid process invocation sequence* might be.

3.3.1.2.1 Invocation

To determine a *valid process invocation sequence*, this research must first define the phrase *invoke process*. In this research, the statement, 'P1 *invokes* P2' means that the CPU has executed an instruction from P1 and that the executed instruction has the intent of requesting the OS scheduler to place process P2 on the dispatch queue for CPU execution. An *invocation sequence* is an ordered series of process invocations.

3.3.1.2.2 Static Process Invocation

Others have made the distinction between statically and dynamically invoked processes [Feng03] [Kos97]. Statically invoked processes are those that are linked into an application; they are part of the same load module as the invoking process [Hof98] [Sek01] [Wag01]. These systems must have their source code or load module analyzed to determine the *valid process invocation sequences*. Static *process invocation sequences* are not of particular interest to this research. To modify processes within a load module would require in-depth knowledge of the

المنارة للاستشارات
www.manaraa.com

application and probably privileged access to make changes to the load module, so such processes pose limited risk to a system.

3.3.1.2.3 Dynamic Process Invocation

Dynamically invoked processes are processes not linked into the application's load module. These processes are invoked in a number of ways: *explicitly*, *implicitly* and *symbolically*. *Explicitly* invoked processes are invoked using fully qualified directory information. *Explicit* process invocations cannot mistakenly invoke a process from the wrong directory. *Implicit* process invocations use the "home" directory. However, the "home" directory actually consists of a number of subdirectories, and the invoked process can be loaded from any one of these. If a number of processes with the same name reside in multiple directories in the "home" directory, the first found process with the name of the invoked process is used, regardless of which part of the "home" directory it resides. *Implicit* invocation using the "home" directory can be dubious at best, because the "home" directory can be changed dynamically.

Symbolic invocations appear to invoke a process from one directory when in fact they are actually invoking a process in another directory. Determining the directory where a *symbolically* invoked process actually resides can be accomplished, although it is not as straight-forward as determining where other external invocation processes resided. *Symbolic* links are more confusing than they are problematic because their target process can be uniquely identified. This research is interested in dynamically invoked processes, because they provide an opportunity to subtly compromise a system.

3.3.1.3 Accountability

A security system requires a means to enable accountability. When a security-related incident occurs, administrators must be able to determine its cause. If a process invoking

المنارة للاستشارات
www.manaraa.com

sequence has not been granted authority to execute, the security system must be able to identify the reason for this situation. At a minimum the system should provide the administrator with the *invalid invocation sequence* and its point of failure. The process invoking sequence can be analyzed to determine the cause of the problem or to determine if the incident represented a false positive result (that is, a result in which a valid process invocation sequence was identified as invalid.) If a false positive result occurs, the security system should be corrected to allow the process invoking sequence to be authorized in subsequent occurrences. In other words, the security system must have the ability to report on the process invoking sequences that it encounters and report which are valid and which are invalid.

Auditors need to review the security system as well, to verify that it is properly protecting the rest of the system, and they need to verify what process invocation sequences are authorized. Assurance that a computer system is adequately protected is essential to meeting government certifications such as FISMA [FISM02] [FISM08]. FISMA certification is mandatory for U.S. government computer systems. The capability to report on the process invocation sequences is an essential function of accountability.

المنارة للاستشارات
www.manaraa.com

Chapter 4: Security Background

As stated earlier, the purpose of this research is to increase operating system (OS) security by detecting and enforcing trusted process invocation sequences within the system. The system must run as expected before any other security measures can be enforced. Without good operating system integrity, any other attempt to secure data, resource, users, etc. is a hollow exercise. If the processes on the system are not those intended to run on the system or if the system is able to run processes in an unauthorized sequence, then the system is already compromised and cannot be trusted.

Operating system security has been an issue since the 1970s, [SCC70] [CST72] [Schr74-1] [Schr74-2] and has become even more so with the increased prevalence of malware and societal dependence upon computers. When referring to operating systems, often security is an after-thought. Operating system designers have focused on the functionality of the OS, not its integrity. [Bish03] [Ravi04] [SHA99] [Smal01a] [Spen99] Various projects have undertaken the challenge of creating an OS with built in integrity [ACM99] [Smal01b]. Commercial OSs has continued to enhance existing OSs by adding or modifying security features in order to assist in maintaining system integrity [ACF99] [RACF03]. And there is a plethora of security add-on tools to further enhance the assurance of system integrity [Appf04] [Ford97] [Mcca05] [Syma05]. All of these approaches have met with various levels of success. This research focuses on one neglected aspect of the system integrity problem, the problem of detecting invalid process invocation sequences and preventing their execution.

4.1 The Program Pathing Problem

Why focus on process invocation sequences? The basic function of a computer is to run processes and, as the computer continues to operate, to invoke other process. To ensure system

المنارة للاستشارات
www.manaraa.com

integrity, it is important to verify that processes run in a valid sequence. When a process runs out of sequence, the system is no longer operating as expected – and therefore can no longer be "trusted." A *process invocation sequence* is the order in which processes are invoked in a system. When the OS boots up, a single process is loaded and invokes a series of other processes until the entire OS is loaded. The scheduler, memory manager, I/O subsystem, and all system services are inter-linked through a series of process invocation sequences. Whenever a user starts an application, another substring of the process invocation sequence is started. Proper operation of an OS consists of only certain process invocation sequences being executed. These process invocation sequences are referred to in this research as *program paths*. When an OS or application deviates from a valid program path (PP), the integrity of that system has been compromised [Schr74-1].

Previous methods to map PP use traces, compiled languages, and even coding PP sequences into programs [ACF99] [Mcca05]. These approaches soon become too difficult and too tedious to administer [ACF99] [Mcca05]. Administrators needed in-depth system knowledge and in some cases had to manually write sophisticated program languages to create new PP mappings. Such mappings either began to take up too much memory, or took too long to calculate [Schr74-1]. As some sparser, more manageable, PP maps began to be used on systems, administrators found deficiencies in their function. These systems could only determine process invocation sequences that were explicitly learned – they could not deduce implied process invocation sequences.

Processes have a number of behavioral characteristics that make the mapping of process invocation sequences particularly difficult. A running process has a unique process identity and location, and may exhibit a number of behaviors such as accessing resources, running privileged

المنارة للاستشارات
www.manaraa.com

instructions, and requesting allocation of executable and data memory. These and other behavioral characteristics, although important for other considerations, may be added to the PP model in later research. This research restricts itself to those characteristics that are relevant to mapping process invocation sequences. The PP model is intended as a fundamental building block to which other behavioral system characteristics can be attached.

Earlier attempts to map trusted program paths and identify some of the pitfalls encountered are discussed in chapter 5. It is the not purpose of this research, to come up with a revolutionary approach, but to keep the solution focused on solving only the process invocation mapping and validation problem. Particular focus is given to the computational theory behind the program pathing solution approach presented (see chapter 6). This research focuses upon an approach that includes the necessary computational power to solve the problem, but no additional computational power.

Before describing the proposed Program Pathing Trust (PPT) Model, it is important to understand the arena in which it participates. Without a background in computer security, the impact of the PPT model may not be obvious. Therefore, this paper first frames the context of computer security in which the PPT model is relevant. The PPT model is not intended as a comprehensive approach to computer security, but as an added dimension to existing security systems. Over the past three decades, the developing discipline of computer security has matured, but has taken many tangents. What started out as enforcement of access control has developed into various other disciplines such as threat management, compliance, security policy and forensics [Harr03]. This research demonstrates that an automaton provides an appropriate computational model to solve an important system integrity problem in computer security, the identification of valid process invocation sequences.

المنارة للاستشارات
www.manaraa.com

4.2 Security and Program Pathing

4.2.1 Detection and Protection Systems

Computer security approaches can be divided into two distinct categories – detection and protection. The detection branch of computer security developed as a result of computers becoming more accessible through the Internet [Harr03]. Public users had direct access to applications and the computers hosting these applications, making it possible for users to interfere with normal computer functions. By virtue of being available to users, systems were no longer isolated and became more vulnerable to a long list of threats: viruses, Trojan horses, worms, time bombs and other malware. Intrusion detection systems were developed to identify the infiltration of these threats [Harr03]. Such systems are traditionally signature based. That is, they can only detect malware intrusions that are known and they are configured to identify [Appf05] [Syma05]. As malware becomes more polymorphic and adaptable, more research is being done on developing detection systems that can identify malware that has not been previously encountered [Kole05]. Research on intrusion detection has relevance to this research and is reviewed in chapter 5.

The protection branch of computer security, about 20 years older, is concerned with regulating access to host computers and their applications. Initially the threat population for computers was limited to the small group of operators and system programmers who had access to computers, isolated in secured data centers. So, much of the attention to security was based on limiting and defending computer resources against unauthorized internal access. Such protection is known as access control and is primarily composed of authentication and authorization.

Access control systems are at the core of a "trusted" system. A trusted system is one that can be relied upon to maintain its status as an uncompromised and reliable system. The concepts of

المنارة للاستشارات
www.manaraa.com

access control and trusted systems have been tightly linked from the beginnings of computer security in the 1970s. They continue to be issues with the prevalence of malware and society's interdependence upon computer applications. In the 1960's and 70's the problem was dealt with by isolating computer systems in secure data centers, and physically restricting access. But even then, there was concern that these systems might not be defendable against internal threats.

4.2.2 Aspects of Access Control Systems (Protection System)

Information security systems can be categorized into two types: data security and system security. This is not a distinction that has always been made, because originally all information security systems either presupposed system security or both data and system security were integrated into access control systems. As the discipline of information security matured, the distinction became clearer; government agencies began to define "trusted" systems and the private sector developed commercial products that aided in implementing "trusted" system integrity, using techniques such as anti-virus, compliance monitors and intrusion detection systems.

Data security systems protect vital information stored and processed by the computer from unauthorized access. System security systems protect the computer's resources and processes. System security is a prerequisite for good data security. An access control security system requires two preconditions: (1) the operating system must have integrity (initially free from vulnerabilities), and (2) the operating system must be protected by a security system (to maintain integrity). The security system and the operating system form a mutually enabling and dependent relationship. The security system can permit damage to the operating system by allowing exposures, malware or malicious users into privileged areas of the operating system. The operating system can circumvent an access control security system by not implementing the

المنارة للاستشارات
www.manaraa.com

proper security intercepts, or by not properly enforcing the privileges required by system programs or personnel, enabling them to disable the security system.

4.2.2.1 Data Security

Data security protects the data being stored and processed by the computer. Data security protects the integrity of the user's data, but can be extended to encompass all kinds of data, files, directories, user applications and computer resources, such as printers, internet access and executable programs. Data security is the ultimate goal of any computer security system.

Data security has been implemented in most operating systems in a variety of ways, but generally as a discretionary access control (DAC) system which allows data owners to grant access as they see fit. DAC systems traditionally require the defining of subjects (users) and objects (computer resources). Subjects are granted access to objects through an enforcement mechanism. Subjects can be granted READ, WRITE, ALLOCATE or EXECUTE privileges to an object. Subjects can also be designated as owners of objects and possibly grant other subjects access to objects they own. [ACF99]

Mandatory access control (MAC) also provides data security, although it is enforced by the operating system using policies and only security administrators have the ability to grant access to objects. MAC systems traditionally are multi-level security (MLS) systems tightly integrated into the operating system. The U.S. Department of Defense Trusted Computer Security Evaluation Criteria (TCSEC), also referred to as the *DoD Orange Book*, defines mandatory security as being associated with security labels (security attributes) associated with objects to reflect their level of sensitivity; security labels are also assigned to subjects [DoD85]. Under MAC, subject and object labels must match or the subject label must dominate (be of a higher authority than) the object's security label for the subject to have access to the object.

المنارة للاستشارات
www.manaraa.com

4.2.2.2 System Security

System security ensures the integrity of the operating system. Traditionally, the primary design goal for operating systems is functionality, not security. It is often said that security is an after-thought in the design of operating systems [AFM99]. As described above, early attempts at system security consisted of simply removing the entire system to a physical environment where penetrability was acceptably minimized. In the 1970s, with the arrival of interactive systems (timesharing, multi-programming, on-line, and multi-processing), securing the operating system became the primary focus of computer security [Schr74-1]. Physically isolating the computer and its access was no longer sufficient as a security strategy because the computer was being accessed by a larger population, sometimes remotely connected by a private network (SNA or LAN). Access control systems, although primarily data security controls, were modified to provide some system security controls.

As computer access extended beyond secure data centers across the world through the internet, maintaining a computer's "trust" status became an imperative challenge for computers running critical applications. In the context of computer systems, "trust" has taken on many meanings over the past decades [DoD85]. This research focuses on the question: can the process invocations in "trusted" computer systems be validated? Both discretionary access control (DAC) and mandatory access control (MAC) contain rudiments of this form of system security [ACF99] [Clar87]. Both protect the system executable files and directories.

المنارة للاستشارات
www.manaraa.com

Chapter 5: Program Pathing Background

5.1 Trusted System

One can easily imagine the failure of a nation's financial system, regional power grid or internet because of cyber terrorism, or accidental interference. A nation's central bank's applications, for instance, may run on a collection of servers, any one of which might process trillions of dollars a day and, if compromised, could cause catastrophic events. Interference to one of these applications could damage the economy, reduce public confidence in the money supply and possibly cause the devaluation of the nation's currency. Therefore, it is important for computers running these applications to maintain a "trusted" status. A "trusted" system is not just free from malware; it also has only prescribed applications running on the system. "Trusted" machines are computers dedicated to run critical applications without interference. Other applications, authorized to run on other systems, may not be authorized to run on the trusted system. This restriction is required because untrusted (or unauthorized) process may steal CPU cycles, reduce performance, or create exposures, causing vulnerabilities and lead to system compromise. An application may be so critical that it is imperative it run without interference – and therefore it must run only on trusted computer systems.

5.1.1 What is a Trusted System?

What constitutes a "trusted" system? A "trusted" system is defined by this research as a dedicated system that is certified to run a critical application and that runs only those processes necessary to support the critical application. "Trusted" systems are required to be locked down with the highest security requirements in order to ensure that the operating system, the applications and the security systems maintain their integrity.

المنارة للاستشارات
www.manaraa.com

Critical applications, as defined by this research, are any applications that are essential to an organization. As a rule, these critical applications are necessary to fulfill the organization's mission; without them the organization would fail. Critical applications have a requirement for high availability and resilience. Examples of critical applications might be associated with nuclear power plant operations, military infrastructure support, central financial applications, life sustaining medical applications, communication systems and navigation systems. The need for a "trusted" system is a function of the organization's tolerance for doing without the critical application(s). If an organization determines that it can do without an application for a period of time, even when failover systems fail, and are willing to accept the risk, then the application is probably not critical and a "trusted" system is not necessary.

5.2 Program Pathing as part of a Trusted System

Program Pathing is by no means the whole solution to the system integrity problem and by itself does not guarantee a system is "trusted." There are many aspects of a trusted system the program pathing model does not address. However, it is an essential part of the solution.

5.2.1 Conceptual Security Models Related to Program Pathing

A number of computer security models have been developed over the past 30 years. Although they are all important, a specific few provide a good background to this research and have an influence upon it.

5.2.1.1 Goguen-Meseguer Model

Goguen-Meseguer took the military lattice approach to information security and created a model to define a "security policy". They make a distinction between security policy and security model. By their definitions a *security model* is a description of a security system,

المنارة للاستشارات
www.manaraa.com

whereas a *security policy* is the set of requirements for a security system. The approach identifies the need to administer a *security policy* that is not static. [Gogu82]

The concept of security policy allowed for the definition of policies such as multi-level security (MLS), capability passing, confinement, compartmentalization, discretionary access, authorization chains and downgrading. The existing concept of "trusted processes" was not that the processes were restricted from running, but that they were restricted from access to sensitive data. Most operating system processes were considered "trusted processes" because they needed universal access to all resources. Goguen-Meseguer considered "trusted processes" such as these to be unnecessarily dangerous, since they could perform any action upon any of the system resources. Their model intended to define precise *security policies* for subsystems by creating domains, and hence restricting the access of processes to resources.

The Goguen-Meseguer model is important because it introduced two concepts. First, it introduced the use of an automaton to model a security solution. The present research goes further and actually uses automaton theory to implement a solution. Secondly, the Goguen-Meseguer model introduced the concept of compartmentalization to security policy, with regards to operating system integrity. This concept arises numerous times in solutions proposed for the system integrity problem. The PPT model restricts valid sequences to only specific authorization paths.

5.2.1.2 Clark-Wilson Integrity Model

Clark-Wilson recognized that the traditional military model of computer security proposed in academic circles at the time was not well suited to the commercial realm. In the mid to late 1980s the military was focused upon mandatory access control systems whereas commercial systems were focused upon discretionary access control systems. The Clark-Wilson model

المنارة للاستشارات
www.manaraa.com

noted the fact that the problem of data integrity existed for both military and commercial environments. The goal of the Clark-Wilson model was to ensure that no user, not even an authorized one, should be permitted to corrupt data, either by accident or with the intent to commit fraud or to be malicious. To this end, Clark-Wilson's model focused upon two concepts: (1) the *well-formed transaction*, and (2) *separation of duties*. [Clar87]

The well-formed transaction stipulates,

> *…that a user should not manipulate data arbitrarily, but only in constrained ways that preserve or ensure the integrity of the data. A very common mechanism in well-formed transactions is to record all data modification in a log so that actions can be audited later.* [Clar87]

In other words, a user may have access to a resource only indirectly through a particular program (or set of programs), written specifically for manipulating the data. Giving access only to the program, without identifying the user would not be sufficient, as "individual accountability" would be lost; it would be known that the program modified the data, but who used the program to modify the data would not be known.

The Clark-Wilson model sets up a data integrity problem to which the program pathing trust model is a solution. That is, the Clark-Wilson model presupposes that all application developers code their applications with this security concept in mind. However, not only do some application programmers not use the Clark-Wilson model, it has an inherent flaw. It assumes that if the program that has access to the data and the user is valid, then the transaction is a "*well-formed transaction*." It is possible, however, for a malicious user or process to invoke the valid process out of sequence of the intended application, thereby circumventing *the well-formed transaction*.

المنارة للاستشارات
www.manaraa.com

5.2.1.3 Brewer-Nash Model (Chinese Wall)

Although Clark-Wilson made mandatory access control more palatable, commercial mandatory access control gained few footholds in the commercial world except for those industries required to comply with government contracts. ACF2®[4] and RACF®[5] both implemented mandatory access control using the Clark-Wilson model, but the feature was rarely used [ACF99] [RACF03].

The Brewer-Nash model integrated the concepts of Clark-Wilson by creating another variation of a mandatory access control. The Brewer-Nash model, commonly known as the Chinese Wall Security Policy [Bre89], defines a model based upon the concept of "conflict of interest classes." The concept is built upon the theory that as a subject gains access rights in one class of data, it restricts the subject's access to other data within the same class. The idea is to keep commercial subjects from profiting "inside knowledge" of data accessed in one area or from gaining similar knowledge from another entity within the same class.

The model is best explained using the Brewer-Nash example of 3 companies. Say that a system stores information on *Bank-company-A*, *Oil-company-B*, and *Oil-company-C*. The model has three levels of "significance" (1) *objects* at the lowest level, (2) groups of all *objects* that belong to an organization or company, the *company dataset*, and (3) the group of all the *company datasets* whose companies are in competition, the *conflict of interest class*. [Bre89] In this example, if a user has access to *Oil-company-B*, the user can be permitted access to *Bank-company-A*, but not *Oil-company-C*. This is because *Oil-company-B* and *Oil-company-C* are in the same *conflict of interest class*, whereas *Bank-company-A* is not in the same *conflict of interest class* as the two oil companies.

[4] ACF2® is an access control product designed for MVS by Computer Associates
[5] RACF® is an access control product designed for MVS by IBM

www.manaraa.com

The Chinese Wall Security Policy model is important to consider when analyzing system integrity, in that it addresses an important problem in multi-processing computer environments. PPT uses Chinese Wall concepts to solve a system integrity problem, in a computing environment where it is not valid to run two different process invocation sequences at the same time. For instance, it may be valid to execute Application A, except when Application B is executing. Running both applications simultaneously may create an integrity exposure – as in the case of running a maintenance process while production processing is running.

5.2.2 Other Implementations of a Trusted System Using Invocation Sequences

The first attempt to define "trusted" system began in December of 1972, at the interim IBM SHARE [Schr74-1] (user group) meeting in San Diego. The SHARE VS/OS Security and Data Management Project met in open session to begin its investigation into the lack of system integrity and computer security in IBM's OS/MVT operating system. It was one of the first known assemblies of computer industry professionals to come together to discuss the topic of creating a commercial computer security system. A diverse group, representing educational institutions, service bureaus, the Department of Defense and commercial industry, met to discuss the requirements for making computer systems secure. Barry Schrager, Data Center Director at the University of Illinois and SHARE Project Manger, documented the findings and requirements of the group in a white paper and presented it to IBM [Schr74-2].

The SHARE Security and Data Management Project focused on two basic concepts: (1) a security system was needed, and (2) as a prerequisite, the operating system had to ensure basic system integrity. SHARE defined system integrity as "*the ability of the system to protect itself against unauthorized user access to the extent that the security controls cannot be compromised* [Sch74-1]." The group identified that these two issues, system integrity and data security, were

المنارة للاستشارات
www.manaraa.com

crucial to a computer security system. The committee's white paper highlighted specific requirements:

- The security system should be an integral part of the operating system
- Identification and validation of users is the first level of security
- The security system should not be able to be turned off
- The system should not have to purge all jobs just to run secure jobs
- The security system should be able to selectively run high-overhead functions on an individual basis.
- A program interface should be the only way to access specific data [Sch74-1].

The last point defines the need for an access control system where data can only be accessed through specified program interfaces (this pre-dates the Clark-Wilson model by 12 years). This requirement was defined by the VS/OS group SHARE requirement #73-86:

> Description:
> There should be a centralized bank of resource control information and an installation replaceable operating system provided service for accessing and maintaining it. The resource control information must relate resources (such as datasets, program paths, etc.), conditions under which they can be made available (such as levels of validation), and user identifiers must all be installation definable.
>
> All authorization and delegation must flow through the single operating system access and maintenance service, and this service must be invokable during normal production operation. Invocation for the purpose of validating access to a resource should return a yes or no answer and optionally a variable length byte string to be used in corrective action (e.g. an error message, module name, or a limit on a quantitative resource). [Sch74-1]

Further discussion identified the issue of validation of the program path in accessing data. Although the technologies have changed since the 1970's, the concerns about program structures providing increased integrity remain valid today. This requirement is at the heart of this research and was specified in VS/OS group SHARE requirement #73-89:

> Description:
> There should be the capability of associating with any dataset a single interface program capable of accessing that dataset. Where the interface program is a subsystem (e.g. IMS) an interface should be provided to other subsystems (e.g. TSO).

المنارة للاستشارات
www.manaraa.com

> Incentive:
> The need to be able to limit the path to a dataset to one interface program structures the system so as to provide increased integrity, security and backup capabilities. [Sch74-1]

In the SHARE security white paper, the program pathing requirement expressed not only a concern for data security but also for data integrity. Data accuracy and completeness, as well as the protection of the data from unauthorized destruction, modification or disclosure (accidental or intentional) were a concern. The requirement recognized that granting access permissions to data was not sufficient for some data, the security system must also identify and validate the interface program structure by which the data is accessed.

In the final IBM white paper, the user group described program pathing as an integral part of the security system as follows:

> Its interfaces to the system should be modifiable so that, with simulation, its decision making processes could be more easily tested, understood and verified. With a well planned set of interfaces via the system control program, it could be easy to use for application programs. Since it would be removed from the application programs themselves, application programmers need not know the exact decision making process that would be used. Conversely, the decision process could be easily modified without having to modify each of the application programs. And finally, since it would easily be removed from the physical resource control, it could easily control conceptual resource such as program paths.
>
> Program paths are transactions, command sequences, and operating processes such as "OPEN". A program path can also be defined to include the flow of control within a module. This enables an installation to define different security levels for different paths within an application program, without having to rewrite different application programs due to the differing requirements for security. [Sch74-2]

IBM response to the SHARE white paper was mixed. They accepted the basic premises, but rejected (or ignored) some of its requirements. The OS integrity requirements were accepted and implemented in OS/MVS. Protection keys and separation of user applications were enforced using virtual storage address spaces. The data security requirements were responded to by IBM with the introduction of their access control product RACF®. However, IBM did not include all the security requirements from the white paper in their newly developed security system, RACF® and program pathing was one of those features missing in the new security system. In response

المنارة للاستشارات
www.manaraa.com

to IBM's rejection of the security model described in the SHARE white paper, a few members of the SHARE group developed a security system using the SHARE white paper as the conceptual design. The result was ACF2®, IBM's primary competitor in computer security [Sch74-2].

ACF2® (Access Control Facility – Second Generation) implemented a version of program pathing. ACF2® was created by SKK, Inc. (Schrager, Klemens, and Krueger, Inc., 1978-1986) in Chicago, Illinois after the founders left the University of Illinois Circle Campus, where the first generation of the ACF® security system was developed

The RACF® philosophy of computer security was opposite of that of ACF2®. Whereas ACF2®'s view of data security was from the resources point of view (rule based), RACF® took an end-user's point of view (profile based). ACF2® based its philosophy on the notion that resources (information) were corporate assets. An organizations main goal was to protect those assets and therefore would want to look at the computer security from that perspective. RACF®, on the other hand, viewed computer security as a means to control user access to assets. RACF took a programmer's or user's point of view. One other primary difference between the two security systems at the time was that ACF2® enforced security by default – access was denied unless explicitly granted. RACF® would later adopt the same strategy.

In 1986, ACF2® was purchased by UCCEL and then the following year by Computer Associates (CA). CA struggled with the program pathing feature in ACF2® as the z/OS computing environments became more complex. Finally, in 1999 CA removed the original program pathing feature and now verifies only the program accessing the resource, instead of the entire program path [ACF99].

The ACF2® version of program pathing is an early version and inspiration of the Program Pathing Trust Model described in this document. It is the purpose of this research to overcome

المنارة للاستشارات
www.manaraa.com

some of the implementation problems that forced CA to remove the feature from ACF2®, and to show how, by expanding the model, it can be used to deal with trusted computer environment problems.

5.2.2.1 ACF2®

ACF2®'s version of program pathing was implemented in 1974, and immediately experienced problems with properly mapping process invocation sequences. In 1999, ACF2® disabled the program sequence checking feature of program pathing, due to the complexity of identifying program paths [ACF99]. The approach of implementing program pathing used by ACF2® was to take a core dump of a running process. The systems programmer would then read the core dump, find the Task Control Block (TCB) and follow the Request Block (RB) chain which represented the program invocation sequence that was recorded by the operating system scheduler. The program path that was discovered in the operating system's TCB/RB chain was then manually translated into assembler MACROs (created by ACF2® developers for that purpose) and assembled into the ACF99@RB module (see Appendix C). ACF99@RB was able to define a number of programming environments using the TCB/RB linkage chains. ACF2® program pathing was not able to use source to develop the mappings in ACF99@RB, due to the fact that not all vendors supplied the source code to their processes.

Figure 5-1 is a representation of a program path making up application 1 with a process invocation sequence of Program1, Program2, Program3, OSProgram1, OSProgram2, and OSProgram3. Program3 requests OS services to OPEN and READ the file. OSProgram3 does the actual OPEN and READing of the data file. Under the ACF2® model of program pathing, the operating system is considered trusted and the ACF2® path does not extend into the operating system program flow. In the ACF2® model only the program state RB chain would have to be

www.manaraa.com

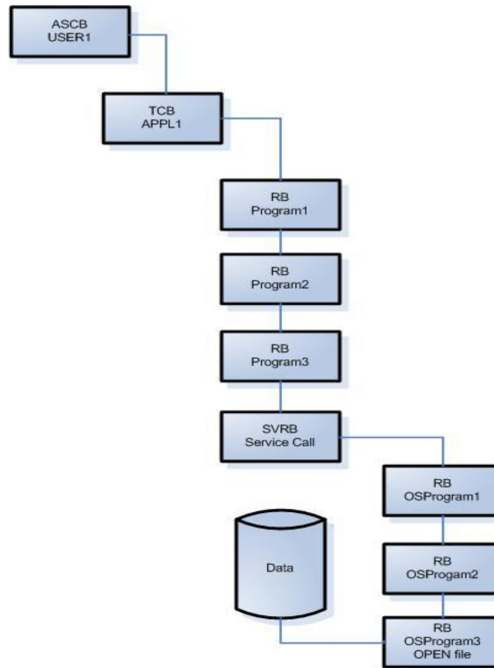

Figure 5-1 MVS Control Block Structure that ACF2 Program Pathing Maps

defined in ACF99@RB. Therefore, only the "Program" program flow would be relevant in the program path. ACF2®'s constructs were not always strictly adhered to from release to release. This was due to the changing architecture of OS/MVS as IBM tightened up system integrity. In some releases, selected parts of the OS programs were often identified in the program path as well. Below is how the example in figure 5-1 would have been coded in ACF99@RB [ACF89].

```
APPL1     @CMD   ,
@TCB   #APPL1

#PROGRAM1       @RB   PROGRAM1,NOSYSLIB,CMD=(CDE,NEXTTCB),LASTTCB,
NEXT=(RB,#PGM##)
#PGM##          @RB  PROGRAM**,NOSYSLIB,CMD=(CDE,NEXTTCB),LASTTCB,
NEXT=(RB,#PGM##,#OS##)
#OS##     @RB    PROGRAM**,NOSYSLIB,CMD=(CDE,NEXTTCB),END
```

[ACF89]

Updating ACF99@RB was a tedious task and took intimate knowledge of the task scheduling subsystem in the IBM MVS operating system. With the increasing complexity of the MVS operating system (as IBM updated MVS with new architectures, going from MVS, to XA, to ESA to z/OS) and fewer and fewer technicians understanding MVS and ACF2 internals,

المنارة للاستشارات
www.manaraa.com

Computer Associates was forced to simplify the program pathing feature in CA-ACF2®. Currently, ACF2® program pathing involves validating only the program that actually issues the OPEN for a dataset (file), in this case, Program3, not the entire program path RB chain (programming environment).

In the current implementation of program pathing, the security administrator only needs to specify the name of the program that issues the OPEN of the file on the ACF2 rule as follows:

```
MY.DATA     UID(*******userid)    PROGRAM(XYZ)   READ(ALLOW)
```

The above ACF2® rule line specifies that the userid can only read the MY.DATA file if the user accesses the file using the XYZ program through the paths defined in ACF99@RB. By dropping program pathing from ACF2®, program XYZ can be validated for accessing the resource MY.DATA, but the paths in ACF99@RB are no long part of the authorization criteria. This presents a problem if XYZ is a generic read program that any programmer can invoke from any program. It may be necessary to validate that a user is attempting access by a program that is authorized to invoke XYZ. ACF99@RB is no longer used in ACF2®.

5.2.2.2 RACF® PADS

RACF®, IBM's z/OS (MVS) security system, implemented program control using Program Access to Data Support (PADS) [RACF03]. PADS performs somewhat like ACF2® in its current implementation. Trusted programs are registered in PROGRAM profiles, and only authorized users can execute these programs (although PADS does not validate the entire program path, it does restrict access to programs). An example of RACF's PADS implementation is as follows:

```
RDEFINE    PROGRAM    XYZ    ADDMEM('SYS1.LINKLIB')
```


المنارة للاستشارات
www.manaraa.com

In the above example, the XYZ program in the "SYS1.LINKLIB" file is identified as a trusted program.

```
PERMIT   XYZ   ID(userid)   ACCESS(EXECUTE)   CLASS(PROGRAM)
```

In this example, the *userid* is given execute access to the XYZ program defined in the previous command.

Data is protected from being accessed by anyone except through a particular program by specifying the program in the data profile:

```
PERMIT 'MY.DATA' ID(userid) WHEN(PROGRAM(XYZ))    ACCESS(READ)
```

The example above specifies that the userid has read access to the MY.DATA file, but only if it is accessed by the XYZ program. PADS performs the same functionality as the current ACF2® implementation.

5.2.2.3 Top Secret®

Computer Associates' other z/OS security product, CA-Top Secret®, defines computer environments with the use of its Facility feature. CA-Top Secret®'s approach is to define the initialization program and the program name id. Under z/OS (MVS), the tradition is that the first 3 characters of a program product are unique to the program product – CA-Top Secret® takes advantage of this to identify a Facility (or programming environment). CA-Top Secret® creates a facility by grouping a set of program names. Taking advantage of the z/OS programming convention that all the program names of a function within an application begin with the same 3 characters, facilities can be defined by masking the program names, e.g., ISP***** (which would define the ISPF programming environment). CA-Top Secret®, however, does not have the concept of program pathing as referred to in this research – it merely names the programs in the program path, but does not identify their invocation sequences. Top Secret® facilities can

المنارة للاستشارات
www.manaraa.com

identify an environment - identify all the programs belonging to an environment, but cannot identify their invocation sequence.

5.2.3 More Recent Background

5.2.3.1 Trusted Path Execution (TPE)

Niki Rahimi (IBM) has done work in the area of program path validation in Linux, taking advantage of the Linux Security Modules (LSM) hooks. His work, the "Trusted Path Execution" (TPE) [Rahi04] although possibly appearing to be similar to the PPT model, takes a different approach. The trusted path that Rahimi refers to in the TPE is the directory path from which an executable resides, not the sequence of program flow path. Although TPE's intent is partially the same as that of the PPT model, to prevent the execution of malicious code, it does not encompass the whole of PPT's strengths. TPE only verifies that a program was loaded from a particular directory.

Rahimi's concept is to validate that the directory paths where a system program resides, and verify that only ***root*** has authority to write to that directory. Any program that resides in a directory that is writeable by any other userid than ***root*** is considered untrusted. Rahimi's TPE is based on the premise that a malicious user can overwrite or damage the operating system code if the directory has the write privilege granted to anyone else but to the ***root*** ID. [Rahi04]

Rahimi's theory works under the assumption that it is a good thing for ***root*** to install all software. However, this does not promote a Role Based Access Control (RBAC) implementation of security. Most security professionals would try to limit the use of ***root*** to only the operating system. Most IT shops are trying to restrict the use of ***root***. ***Root*** has "all powerful" authority – the user using ***root***, cannot be identified when s/he performs activities, making individual accountability difficult. In addition, there is no good way to restrict a user's

المنارة للاستشارات
www.manaraa.com

use of root to only those tasks s/he needs to perform. SUDO [Mann03] (under UNIX and Linux) could be used to restrict a user's use of root, although SUDO has flaws that enable a sophisticated user to get around these restrictions.

The goal of a true RBAC implemented system is to (1) restrict all users to only those functions and data they are required to perform, and (2) to log a user's actions on the computer, so as to ensure individual accountability. If all software had to be installed with ***root*** then too many technicians would have to be granted ***root*** authority, because in large shops there are multiple technical roles. TPE provides a good mechanism to ensure that a program is coming from a directory in which it was installed, however the theory needs further refinement if it were to fit an RBAC implementation of security.

The PPT model can be used in tandem with Rahimi's TPE; they are not incompatible. PPT deals only with the progression of the process invocation sequences, not with validating the directory from which the program resides, so this is an aspect of TPE that would enhance a PPT implementation (see section 9.3).

5.2.3.2 Symantec's Critical Program System (CPS)

In Symantec Corporation's purchase of Platform Logic, it acquired the Host Intrusion Prevention product, AppFire® [Appf04]. AppFire®, now enhanced and re-branded Critical System Protection® (CPS), approaches system integrity using a behavior-based approach. Symantec's approach is based upon the concept that each software program accesses particular resources and accesses them in a particular way. For example, a program may have to create, update or read a log file, or access a particular tablespace in a database. Behaviors might be described as such things as functions of the operating system or application as it accesses files, registries, devices, network connections or other system services.

المنارة للاستشارات
www.manaraa.com

Behaviors are defined in Behavior Control Descriptions (BCD). Each BCD is a set of behavior definitions defining a set of resource names, access permissions requested and time or frequency of access. The BCD is in turn associated with a set of processes or a logical group of processes invoked a Process Set (PSET). The PSET associates a set of resources and permissions to the set of processes defined in the PSET. The Process Binding Rules (PBR) assigns a process to a process set (PSET).

BCDs are defined with the product's "profiler tool," which can be set to automatically generate a BCD. The process of automatically creating the BCD is referred to as "self-learning". One of the chief advantages of CPS is that the self-learning mode provides a "crystal box" approach, where the administrator can audit the behavior controls generated, and the administrator can review and modify the generated behavior policy defined in the BCD.

CPS's architecture is not concerned with program pathing (program flow control), however it does offer an interesting self-learning concept using the "crystal box" technique. Unlike many "self-learning" systems, CPS provides the administrator with the ability to review what the system has learned in human-readable format, providing the administrator the opportunity to fine-tune and correct the BCD access permissions. The concept used in CPS more closely resembles the Clark-Wilson model implemented by ACF2®'s program control and RACF®'s PADS than it resembles PPT.

5.2.3.3 SELINUX

Security-Enhanced Linux (SELinux) [NSA01] [Mcca05] [Smal01b] is a National Security Agency (NSA) project created to protect against the exploitation of vulnerabilities in Linux. SELinux is a mandatory access control system developed to secure government systems for critical applications. SELinux is based on Flux Advanced Security Kernel (FLASK) [Spen99], a

www.manaraa.com

security architecture framework for operating systems. The FLASK architecture is based upon the Flux OS toolkit [Ford97] and was prototyped in the Fluke OS [AFM99]. SELinux is built upon a mandatory access control, but it departs from the traditional definition of mandatory access control. Unlike the more traditional versions of mandatory access control (MAC) as defined by Bell-LaPadula [Bell76], Biba [Biba77] and the Clark-Wilson [Clar87] models, the FLASK model is a policy-based model. The basic components of SELinux are a combination of type enforcement (TE), role-based access control (RBAC), and multi-level security (MLS). The policy is made up of a reference policy language. It is compiled and loaded into a reference policy in the Linux kernel.

The reference policy is made up of a policy language that defines computer types. SELinux defines many kinds of types, but a simplistic example is one which defines types of the attributes files and processes. Processes are defined in domains. Files are defined as resources. After defining domains and resources, the reference policy language defines the domains' access to the resources, as shown in figure 5-2.

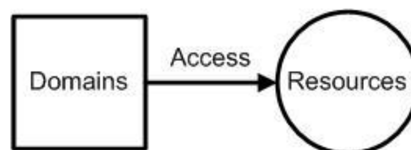

Figure 5-2 SELinux Domain - Resource Concept

Figure 5-3 is an example of a SELinux policy reference language [Mcca05]. Note that the reference policy language defines all the computer entities, both files and processes. The first part of the policy reference language defines the types, then the attributes of

المنارة للاستشارات
www.manaraa.com

```
type    daemon.edit;
type    daemon.d;
type    daemon.log;
type    daemon.conf;
type    port_80;
domain    daemon : { dirFiles piple };
resource    port_80 : { direFiles sockets };
domain    daemon.edit : { dirFiles };
domain    daemon.d : {dirFiles pipes };
resource    daemon.log : { dirFiles };
resource    daemon.conf  : { dirFiles };
access    daemon.d    port_80   read { dirFiles:stat
sockets:read };
access    daemon.d    port_80   write   { dirFiles:none
sockets: read };
```

Figure 5-3 SELinux Policy Reference Language for daemon.te

the types are assigned. In the second part of the policy the domain type is granted access to each resource type. Figure 5-4 illustrates a graphic representation of the security policy in figure 5-3.

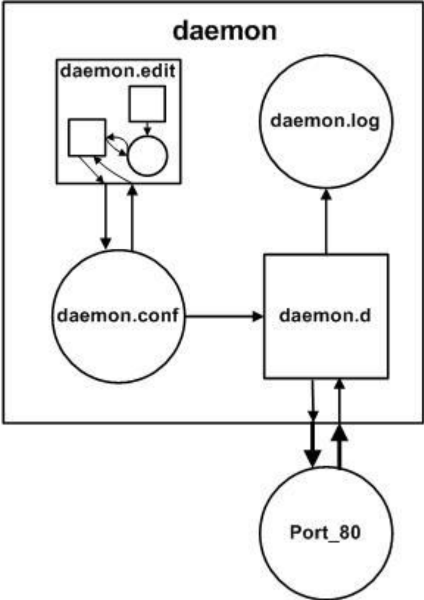

Figure 5-4 Conceptual Diagram of Daemon Policy Reference Example

An interesting feature of SELinux is how it maps process invocation sequences. It groups processes in domains and identifies which processes in the domain can invoke other processes in the domain using type enforcement. SELinux's type enforcement (TE) domain transitions are based upon the association of programs (or processes) within domains. A domain is a set of like-programs (or processes) that work together to create a function (domain). A domain of programs

المنارة للاستشارات
www.manaraa.com

is a type. Each domain is assigned a set of permissions that allow the domain (the set of programs) to perform a function. Domains can invoke other programs within the same domain or in other domains using "transition" rules. SELinux's focus, however, is not restricted to the invoking sequence of one process invocation to another, but on one domain transitioning to another or other types (files, sockets, etc.) – this is determined by type enforcement (TE), which is type transition rules.

The reference policy language illustrated in Figure 5-5 [Macc05] modifies the reference policy in Figure 5-3.

```
type  daemon-init;
type   daemon.edit;
type  daemon.d
domain_type(daemon-init)
init_daemon_domain(daemon-init,daemon.edit,
daemon.d)

allow    daemon-init    daemon.edit:process
transition;
allow    daemon-init    daemon.d:process
transition;
```

Figure 5-5 SELinux Process Control

In this new domain, the daemon.init process is added to the daemon domain and becomes the initialization process that invokes the daemon.edit and daemon.d processes. The daemon domain is modified as shown in Figure 5-6.

The SELinux reference policy language is not easy to use – and the examples above are very simple cases, not using all the capabilities of the language. Although the language is very powerful it requires in-depth knowledge of the processes, files and other resources running on the Linux system in order to use it well. There are literally hundreds of man-years of development of the SELinux reference policy. At this time the kernel reference policy has been

المنارة للاستشارات
www.manaraa.com

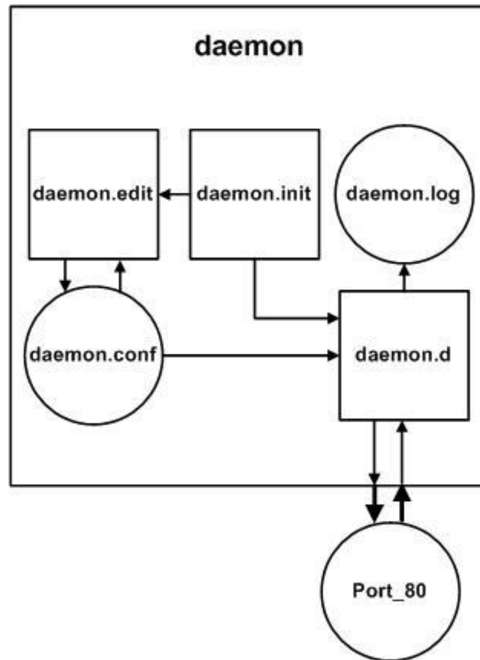

Figure 5-6 SELinux Process Transitions

nearly completed. No application programs have had a reference policy written yet. To write the reference policy for an application would require someone to have both in-depth knowledge of the internals of the application and proficiency with the reference policy language.

5.2.4 Current Literature on Program Pathing

The current literature on process invocation sequence validation is not in the realm of access control, but are approaches intended to solve intrusion detection issues. The literature focuses upon discovering *invalid process invocation sequences*, and does not deal with the prevention of unauthorized processes or processes running out of sequence. Figure 5-7 is a citation map of the literature showing the evolution of process invocation sequence mapping (see section 2.2) in intrusion detection. Although the literature does not specifically address all the requirements this research is investigating, it does deal with the problem of mapping the process invocation sequences (refer to section 4.1) which is central to solving the system integrity problem.

www.manaraa.com

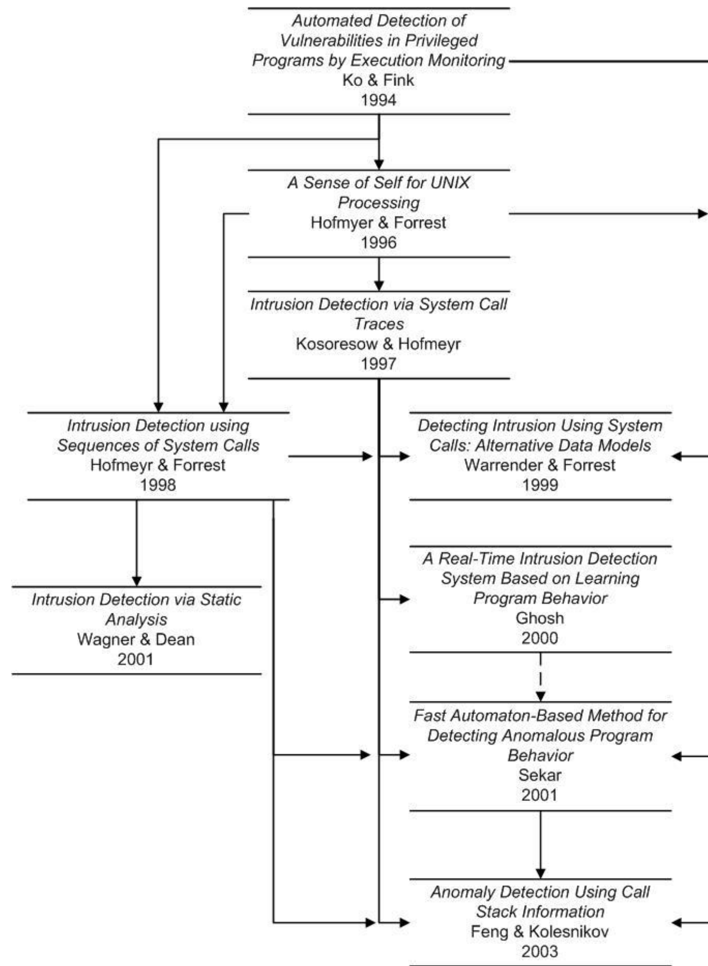

Figure 5-7 Literature Mapping

5.2.4.1 Non-Computational Theory Approaches

As stated earlier, one of the premises of this research is to use computational theory as a basis. A number of approaches in the literature depart from our approach. The next four approaches are examples of some of these unique approaches.

5.2.4.1.1 Hofmeyr-Forrest – N-Gram Approach

Every program produces a set of process invocation sequences. The sequences are determined by the execution order of the processes. Each process is dealt with as a black box – the process invocation sequence does not review the internal workings or role of the process, outside its invocation of other processes. Hofmeyr-Forrest [Hof98] defines these sets of

المنارة للاستشارات
www.manaraa.com

sequences as normal behavior and divides the sequences into pattern lengths of 5, 6, or 11 processes. Patterns allow for a look-ahead expectation of what processes should follow an executed process. The patterns (*n*-grams) are stored in a database as normal behavior sequences. Any behavior not matching one of the *n*-gram patterns is determined to be abnormal indicating an anomaly.

The approach is performed in two stages. The first stage scans traces for normal behavior, where patterns are created and stored in a database. In the second stage new behaviors from traces are matched to patterns in the database. Anomalies, new behaviors that are found to be different than those captured in the *n*-gram patterns, are reported as intrusions. Process invocation is determined whenever a process is created (or invoked) using a *fork* or *vfork*. In Hofmeyr-Forrest's research, only the fork processes were collected as invoked processes. The *vfork* is created in the process invocation sequence as a new process ID, and is therefore difficult to associate in a trace with the process invocation sequence (which has a different process ID). This difficulty is easily overcome by profiling the invocation sequences by intercepting processes before they are placed on the dispatch queue.

Collection of normal behaviors can be accomplished by one of two methods. First, the database can collect learned normal behaviors automatically from traces, through a series of tests. Or normal activity from a running production system can be monitored and the database can collect the learned normal behavior. Secondly, the database can be loaded from manually constructed traces. These are normal variations of possible normal behavior created by the researchers. This latter is the approach taken by Hofmeyr-Forrest to test their approach. It was felt that by using these artificial normal behaviors, more variations in behaviors could be

المنارة للاستشارات
www.manaraa.com

captured and fewer false positives would be detected (i.e. the system would identify fewer false intrusion detections).

Hofmeyr-Forrest identifies a number of hurdles that must be dealt with before the theory can be put into practical use. The hurdles identified involve both operating system issues and problems with the *n*-gram theory that still must be resolved. The operating system issue is that most such systems do not provide the necessary trace facilities with the required detail to collect the process invocation sequences. Therefore, either a better trace facility would have to be provided by the operating systems or one would have to be added to the OS. The issue with the *n*-gram theory is that there is no "stopping criteria." That is to say, there is no criterion by mean of which it can be determined whether the system learned enough different process environments to fully capture the process invocation sequences in a system.

5.2.4.1.2 Warrender - Forrest – Alternate Data Models

Warrender- Forrest [Warr99] uses the hidden Markov model (HMM) to create a structure to map process invocation sequences. The hidden Markov model is a Bayesian network where the state transitions are probabilities. The HMM takes much longer to train than the other approaches discussed in this paper. Warrender- Forrest stated that HMM took two months to train as opposed to other methods that were trainable in a matter of hours for their largest training data.

Another disadvantage of the HMM method is that the researchers had to predict the number of states needed for the number of system invocations. They used a 40-state HMM in most cases. In this approach, prior knowledge of the process invocation environment is needed. The Warrender-Forrest approach is more complicated than needed for detecting invalid process invocations.

المنارة للاستشارات
www.manaraa.com

The HMM is not only more complex than is needed, it also takes more time and effort to train and more computing power than other methods. Therefore, the HMM is not a good candidate for our purposes, since predicting probability of new process invocation sequences is not necessary in validating them.

5.2.4.1.3 Ghosh – ANN Approach

Artificial Neural Networks (ANN) is an attractive approach because it's a relativity simple approach to code and calculate. Ghosh's [Gho00] approach uses a combination of three algorithms to create an ANN supported by a finite state automaton (FSA). The ANN algorithm used is the Elman recurrent neural network [Elm90]. Unlike traditional forward-feed back propagation neural networks, the Elman neural network has a feedback loop from the hidden layer to a context layer, which gives the new input another source of input feedback from the previous input string. In effect, this allows a string to be broken up into smaller substrings. As an input is fed into the input nodes, they are propagated into the hidden nodes, and the results are fed into the output nodes and context nodes. The next input data is fed to the hidden nodes, and the context nodes using the previous input's context node results are fed into the hidden node. The FSA accepts all the sequences from the training data.

The data fed into the Elman ANN is converted into *n-gram* sequences which are further divided up into *l*-gram sequences ($l < n$) by the string transducer. The *l-gram* sequences are fed into the Elman ANN input nodes for training and later for verification. One or more *l-grams* can make up an *n*-gram (see section 5.2.4.1.1). A process invocation sequence can be made up of multiple *n*-grams which are recorded into the FSA by the "state tester." The "state tester" is responsible for automatically creating a FSA to represent *valid process invocation sequences*. The training data is made up only of *normal behavior* and is used to profile *normal behavior*.

المنارة للاستشارات
www.manaraa.com

The FSA's transitions relate to specific *l*-gram sequences, which in turn make up one or more *n*-grams mapping *normal processes*. The *n-grams* and *l-grams* are just substrings of the process invocation sequence. The n-grams serve the same function as those in the Hofmeyr-Forrest approach. The *l-grams* are produced from the *n-grams* so they can be fed into the ANN.

One of the objections to ANNs is that it is difficult to determine what the ANN has learned. The *n*-gram and FSA appear to help resolve this issue. The combination usage of the *n*-gram and Elman ANN solves the problem of determining how many input nodes to use in the ANN. And the use of the Elman ANN creates a good decision making engine. Like most ANN systems, it is difficult to determine accountability. The ANN can identify what *process invocation sequences* are valid, or invalid, but cannot determine why, because the ANN cannot identify specifically what part of the process invocation sequence it found invalid. This is because ANN uses stochastic gradient decent to determine whether or not a sequence is valid or invalid [Mitc97]. The ANN translates the sequence into statistical relationships and the original input is lost. This is the major objection to the ANN approach.

5.2.4.1.4 Ammons -Larus – Retrieval Tree Approach

Although Ammons-Larus's [Amm98] research is concerned with program execution paths as opposed to process invocation sequences, it does illustrate other approaches that can be used to represent process invocation sequences. Mapping program execution paths (internal branches within a load module) involves different kinds of processes than mapping process invocation sequences; however, they both have a similar intent. They both map the program paths of an application or program. Program execution pathing maps the internal processes (see section 2.1.2) whereas process invocation sequences map the external processes (see section 2.1.1) of one process to another. So their research has some relevance to this research.

المنارة للاستشارات
www.manaraa.com

The Ammons-Larus method is an adaptation of the Ball-Larus [Ball96] method. The Ball-Larus approach maps execution paths to a direct acyclic graph (DAG). Program execution paths (DAGs) are profiled in a control-flow graph (CFG), where each edge is labeled with the frequency over a number of dynamic tests. As the test data is run, each execution path is recorded in the CFG and each edge is updated by one as the program's execution path is recorded. Each subsequent program test is captured into a CFG.

The CFG is converted to use a DAG to model the execution paths. They heuristically identify all explicit paths; the DAG optimizes the spanning tree so that no edges sprouting from a node have the same value. This allows the diagram to identify the optimal path. DAGs are appealing because they are used in system scheduling theory to optimize task scheduling of jobs to find the longest and shortest paths.

Ammons/Larus uses Ball-Larus's techniques, where the path profiles count the number of times a program executes acyclic paths in a CFG. Ammon-Larus identify the "hot" paths using the frequency values on the edges. Then using the Aho-Corasick [Aho75] algorithm Ammons-Larus construct a retrieval tree from the hot paths. A single retrieval tree can represent a number of program flows. Note that unlike Ball-Larus, Ammons-Larus repeats patterns as it progresses through the program path. Program flows with the same beginning prefixes are organized together. As paths are validated against the retrieval tree, if a path enters a state where there isn't an edge that matches the path, then a failure function is entered, and the path is reset.

Ammons-Larus claim that if there are no paths that match the substring, then the failure function resets the automaton. However, Ammons-Larus do not define their automaton. It may be that they are only referring to their structure loosely as an automaton and do not mean it as a formal automaton in the theoretical sense. The retrieval tree structure lacks an accepting state.

المنارة للاستشارات
www.manaraa.com

The retrieval tree also does not take advantage of the Keene closure, although this is not a requirement of a deterministic finite state automaton. They divide their paths into partitions, which has advantages if we want to isolate different invocation sequences that may be mutually exclusive. A version of this is accomplished in the PPT model. Operations that are mutually exclusive are defined in separate FSA machines.

5.2.4.2 Computational Theory Approach

This research shows that a finite automata approach is the preferred computational model to use. Other researchers have had similar hypotheses, and their work is reviewed in this section.

5.2.4.2.1 Ko-Fink – Execution Monitoring

Ko and Fink's [Ko94] approach analyzes and maps the behavior of privileged programs as a comparison reference with actual program behavior recorded in system audit logs. Although privileged programs can potentially do anything, they tend to perform intended behavior that is limited and benign. Privileged programs can bypass both mandatory and discretionary access control mechanisms due to their root privilege. Ko-Fink's research developed a language to monitor these privileged program behaviors. The monitoring language verifies that resources can only be accessed through invocation of the proper system invocations.

Unlike other approaches, the mapping of invocation sequences in Ko-Fink mainly focused upon processes executing with privileged access. Unprivileged processes are mapped, but only as they relate to the privileged processes. Any unprivileged process invocations encountered are considered part of the privileged process invocation sequence.

Ko-Fink maps the process invocation sequences using a language based upon predicate logic and regular expressions. It is created manually from system audit logs. The language is made up of three operands described in the example below which define the following: (1) The name of

المنارة للاستشارات
www.manaraa.com

the process executing and the objects it manipulates, (2) the attributes of the processes, or objects, for example the owner or permissions, and (3) the current operations of the program execution, for instance what the program can do. An example of the language follows:

```
#define  mailboxdir          "/usr/spool/mail"
#define  mailport            25
#define  root_mail_handler "/home/root/mail_handler"
PROGRAM sendmail(user)
   read(X)  :-  worldreadable(X);
   write(X)  :-  inside(X, mailboxdir);
   bind(mailport);
   exec("bin/mail");
   exec(root_mail_handler)  :-  user.uid = 0;
END                                        [Ko94]
```

The "#define" operand defines the attributes of the objects in the rule. The "PROGRAM" operand defines the beginning of the rule. The "exec" operand identifies the other processes invoked in the sequence.

Ko-Fink uses a *Sendmail* example to illustrate how the mapping works. The *Sendmail* program performs a number of functions: (1) it runs as a daemon process to accept mail from mail ports and route the mail to remote systems, (2) it executes mail handlers on the user's behalf and (3) it resends pending mail in the mail queue. *Sendmail* has a vulnerability that enables a user to make *Sendmail* execute a user program with root access. The Ko-Fink monitoring language restricts *Sendmail* to executing only processes in the /bin/mail directory (shown in the example monitor language rule above). The monitoring language is a form of process invocation sequence verification – *Sendmail* can only execute processes from a specific directory and not just any program specified by the user.

The merit of this approach is mentioned by other significant papers that investigate alternative approaches to the process invocation sequence problem. The Ko-Fink approach, although not practical from an implementation stand-point, does define some of the issues

المنارة للاستشارات
www.manaraa.com

around why mapping and verifying process invocation sequences is important. It shows a number of vulnerabilities that can be exploited by users to compromise security using some simple techniques, underlying the need for controlling process sequence validation. The problem with Ko-Fink's approach is that mapping the invocation sequences with the language requires it to be done manually by a knowledgeable technician.

5.2.4.2.2 Kosoresow-Hofmeyr – System Call Traces

Kosoresow-Hofmeyr's [Kos97] research is probably one of the most important approaches in the recent literature as far as mapping process invocation sequences is concerned. In this approach, process invocation sequences are mapped from system traces using a regular language and are used to construct a deterministic finite state automaton (DFA). A process invocation sequence is mapped in the form of a regular language construct called a macro. Macros do not map an entire process invocation sequence, but are built from repeatable patterns found in the system calling traces. Macros are created from a three-phase process, (1) a process is executed, which may or may not generate a process invocation sequence (invoke other processes). A trace of system invocations is produced, showing the processes invoked from the initial process. (2) The system call trace is analyzed by a script that identifies the invocation sequences for a particular execution path; this is done to reduce the size of the invocation sequences and identify the common invocation sequence patterns characteristic of the process invocation. The script also identifies only process invocation sequences that are interesting (part of the execution sequence being mapped), because there is always a number of system and other overhead processes running that are not part of the process invocation sequence, for instance system service calls which are not mapped as part of Kowsorsow-Hofmeyr's approach. (3) Finally, a set of macros are created heuristically from the analyzed system calling traces. This is done though

المنارة للاستشارات
www.manaraa.com

a combination of scripts and manual coding of the language. The macros are then loaded into a DFA using another set of scripts.

Each unique invocation sequence is divided into three parts: a prefix, a main body and a suffix. For each invocation sequence the common prefixes and suffixes are identified. Then the main body sequences are scanned for common reoccurring strings of two to six invocations. Each recurring substring of common prefix, shortened main body sequences and suffix become macros. The macros take advantage of Kleene closure [Hopc01] to reduce the size of the process invocation sequences captured, for example when process x invokes itself, x^* is used where x is represented 0 or multiple times. The macros are then loaded into a DFA.

By using this method, the researchers found that they could reduce the number of invocation string instances they had to capture. The technique could take advantage of all the different reoccurring patterns (macros). In one case, the researchers found that instead of mapping 75 strings they could capture the same representation in 36 macros. The approach drastically reduced the size of the process invocation sequence map. And the researchers found that their approach provided "a reasonably close approximation of normal behavior." Their approach admittedly used a combination of scripts and manual methods to create the DFA, and the issue of efficiency was an acknowledged issue by the researchers. They admitted an exact DFA representation of a process invocation sequence was likely to be problematic, mainly because to do so the DFA would have to map every possible system invocation sequence, which would likely cause the DFA to become too large. Secondly, there would be false negatives, because some process invocation sequence variations would probably be missed. Thirdly, the more sequences that were mapped into macros, the longer the calculation time would be to determine

www.manaraa.com

the macros. The program pathing trust model developed in this paper addresses the problems encountered by the Kosoresow-Hofmeyr model.

5.2.4.2.3 Sekar – Finite State Automata Approach

Sekar [Sek01] also uses a finite state automaton to capture invocation sequences. Unlike previous approaches, the automaton approach is not limited by the length of the invocation sequence. Longer, more complicated invocation sequences do not pose a problem and it is computationally efficient. It easily accommodates program loops and branches. The FSA approach also uses the program counter (PC) to capture information about process invocation sequences that would not be possible in the static approaches. The advantages of the FSA approach over previous approaches entails:

- Faster learning – Entire invocation sequences can be learned at once, as opposed to learning multiple *n*-grams. Experimentation shows that convergence occurs quicker in the FSA model than in the *n*-gram model.
- Improved detection – Using the PC enables the FSA to detect classes of attacks that were not detected in other approaches. For instance, the *n*-gram approach can determine that an invocation sequence has executed valid *n*-gram sequences, but it cannot determine that it has missed or skipped a sequence. Also detection takes less computational time and is quicker than other approaches. The FSA has the advantage of being kernel-based, although this is also a disadvantage because it entails modifying the kernel.
- Fewer false positives – The FSA can generalize learned normal behavior, in essence predict unlearned behavior, whereas, the previous approaches can only identify those sequences they have learned.
- Simpler and more compact representation – The FSA takes up less memory. For instance,

المنارة للاستشارات
www.manaraa.com

where the *n*-gram approach needs to represent 51 system invocations the FSA only needs 13.

The FSA is trained in real time. Normal behavior is recorded in the FSA in real time, capturing the system invocation name and the point at which the system invocation was made, (the value of the PC when the invocation was made). Each value in the PC represents a different state in the FSA. The system call name (the name of the process being invoked) is represented by the transitions. Sekar represents the automaton by symbolizing the transitions in pairs $\frac{SysCall}{PC}$ and $\frac{Prev\ SysCall}{Prev\ PC}$ where the transition from state *Prev PC* to *PC* is labeled by *Prev SysCall*. Sekar encountered a problem with dynamically linked programs - because these programs may get loaded into different locations, they cannot be relied upon to have the same PC location from one invocation to another. Another problem is that system function invocations may invoke other processes that cause multiple extensive system invocation sequence branches, which then return control back to the original invocation process. Figure 5-12 represents three invocation sequences:

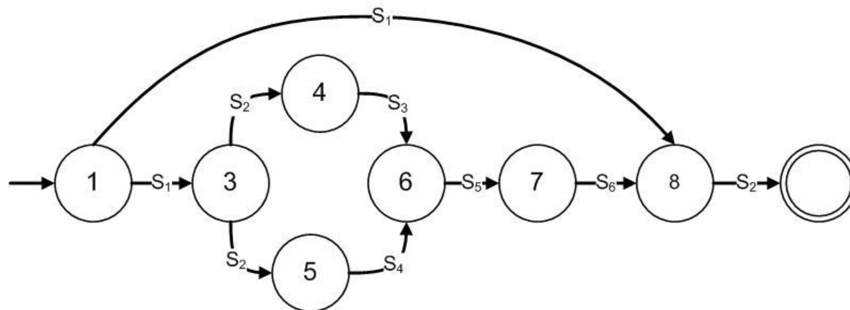

Figure 5-8 Seka's FSAFigure

The invocation sequences in 5-12 are:

$$\frac{S_1}{1}, \frac{S_2}{8}$$

$$\frac{S_1}{1}, \frac{S_2}{3}, \frac{S_3}{4}, \frac{S_5}{6}, \frac{S_6}{7}, \frac{S_2}{8}$$

المنارة للاستشارات
www.manaraa.com

$$\frac{S_1}{1}, \frac{S_2}{3}, \frac{S_4}{5}, \frac{S_5}{6}, \frac{S_6}{7}, \frac{S_2}{8}$$

As processes are invoked by the operating system, they are placed in frames on the process stack. Each process frame represents an invoked process and contains the return address, parameters passed to the process and local variables defined to the process. This is provided in all programming languages.

The *fork* and *exec* system invocations provide a unique problem for Sekar, as they either create another copy of the invocation process or create a child process. The *fork* and *exec* are basic system invocations that appear quite often in invocation sequences. For Sekar's FSA approach, he must decide if processes invoked by a *fork* or *exec* should be recorded in the current FSA or a new FSA – this suggests that Sekar is building an FSA for each process invoked by a command line or deamon.

This model most resembles the PPT model, and chapter 6 shows how they differ. The major difference between Sekar's approach and the Program Pathing Trust Model is that the Sekar method maps process invocation sequences within a load module, requiring either the source code, the op codes for the invocations or hooks in the *exec* and *fork* intercepts to map the invocation sequences. Whereas the Sekar model has problems mapping dynamic invocations, the program pathing model focuses upon the mapping of these invocations.

5.2.4.3 Context Free Grammar or Pushdown Automata

There are a number of process invocation mapping approaches that are based upon theoretical computational models that are not exclusively finite automaton based. The approaches described below are a examples of models that have developed more computationally complex models.

المنارة للاستشارات

www.manaraa.com

5.2.4.3.1 Feng - Kolesnikov –Pushdown Automata Approach

The Feng-Kolesnikov approach uses the system call stack to gather information about *valid process invocation sequence* behavior, and develops a pushdown automaton. Like Wagner and Sekar, Feng [Feng03] uses the system call stack and PC (Program Counter), to gain more information about the nature of invocation sequences. From the PC the current executing program's next instruction can be determined. From the call stack, the process invocation sequence, its status and the invocation program's return address, from which the offset into the invocating process where the invoked process was invoked, can be determined. Feng's approach is called VtPath, and is unique in that it uses the return addresses recorded in the call stack. The method abstracts two execution points, one from the invoking process and the other from the invoked process, and determines if they are valid based upon previously learned normal behavior.

Training is performed by gathering the process return addresses from the call stack and is recorded in a hash table called RA (return address) table. If the return address is the last entry in the virtual stack, then the call number is saved with it. A VP (virtual path) hash table is used to save the parent invocation sequences for the process. During training all *valid process invocations* are added to the hash tables saving their return addresses and virtual paths.

The VtPath approach is able to handle Dynamically Linked Libraries (DLLs), which the Sekar approach had difficulty handling. VtPath claims that the Sekar's FSA approach was an "unnecessary simplification." The VtPath uses call stack history, as well as PC information. By doing so, the VtPath traverses function paths that the FSA method only records the system call name and the current PC. The VtPath also records the return addresses from the call stack in the VP stack. The VP stack represents a history of all the unreturned functions.

المنارة للاستشارات
www.manaraa.com

The approach introduces a pushdown automaton, mainly because it is dealing with the program counter and system call stack. The researchers, dealing with a pushdown automaton implemented in the OS, emulated the same structure to solve their problem of mapping process invocation sequences. However, the problem of mapping and validating process invocation sequences is not the same problem as managing process flows. Validating process invocation sequences does not require the number of times a process is invoked recursively for instance, or involve the fact that if process A is invoked 4 times, then process B must be invoked 4 times. Therefore, a pushdown automaton has more computational power than is needed. This is analyzed in more detail in section 6.2.1.5.

5.2.4.3.2 Wagner- Dean – Pushdown Automata Approach

Wagner-Dean [Wag01] also attempts to solve the problem of detecting system intrusion by profiling *valid process invocation* behavior. Wagner-Dean's assumption is that formal methods alone are insufficient to build a system to model *valid process invocation* behavior. They base this observation on the fact that formal systems have been used for 25 years, and have yet to realize this goal. They agree with the basic premise of these systems, however, that a system's behavior can be learned by observing its normal behavior. The assumption is "*a compromised application cannot cause much harm unless it interacts with the underlying operating system, and those interactions can be readily monitored*"[Wag01].

The Wagner-Dean approach begins by creating a model of the expected behavior of an application from program source code. Then, they monitor the program and check the system call trace for compliance to the model. Although Wagner-Dean uses practical observation, their models do use formal languages, either regular or context-free languages.

المنارة للاستشارات
www.manaraa.com

Wagner-Dean's research investigates four approaches to capturing expected process invocation behavior. The four models used are the trivial, the callgraph, the abstract stack and the digraph models. Each model refines the normal behavior represented a little more to prevent false positives. The trivial model is a regular language model that is inferred from a parse tree. It identifies the set of system invocation from an alphabet S. The model of normal behavior is defined from a regular language S^*. Any invoked process not recognized by the language is an *invalid process invocation*. The trivial model identifies all the processes that are valid, although does not identify the sequence. The model is fed by analyzing source code to determine what process invocations are performed for an application. Wagner-Dean believe that although the trivial phase is "*easy to implement, sound and efficient*", [Wag01] it does not detect attacks that use *valid process invocation sequences*. The approach is not granular enough to detect abuses of the *valid process invocation sequences*, such as the ***open( )*** system invocation, which can be used to modify any file – including another file that is an executable in the language S^*.

The trivial model just identifies the processes that are allowed to be invoked within an invocation sequence, but does not identify the ordering of the invocation sequence. Ordering of the alphabet S is performed by Wagner-Dean using a non-deterministic finite automaton (NDFA). The NDFA (or callgraph) is built by performing a flow-control analysis upon the application The NDFA is built assuming that only one invocation can be made from a single application location. In this NDFA, every correct transition state is considered an accepting state. However, this approach has the problem of showing how processes return control to the originating process. Function calls are a particular problem; they are invoked for services and are not part of a long invocation sequence. They appear to be dead-end nodes on a NDFA. The

المنارة للاستشارات
www.manaraa.com

dead-end nodes (impossible paths) result in a larger than needed NDFA. To deal with the problem of these dead-end nodes, the abstract stack model is introduced.

The abstract stack model is a non-deterministic pushdown automaton (NDPA) which allows for a context free language. It emulates the *program counter* and the call stack in the operating system. The dead-end nodes are not a problem, because they are pushed on the NDPA when they are invoked, i.e. *entry(f)*, and popped off the NDPA when they return control, i.e. *exit(f)*. Again the NDPA is created from an analysis of application source code. However, the use of the NDPA model proves to be a challenge. In monitoring an application's invocation behavior the NDPA has to search for all possibilities, this can be theoretically and computational exhausting.

Another problem that Wagner-Dean discusses is NDPA's difficulty in monitoring activity for intrusions, because of the need for a top-down analysis. A top-down analysis may be needed if intrusion detection is being done against a system trace log. However, if intrusion detection is being done against real-time processing, the bottom up approach that the NDPA lends itself to would be an advantage. If a process is intercepted in the scheduler before it is executed, the system data area representing the process has pointers back to the process that invoked it. That process has a data area with pointers to who invoked it, and so on. Therefore, a bottom up mapping of process invocation sequences would be advantageous.

المنارة للاستشارات
www.manaraa.com

Chapter 6: PPT Theoretical Model

By analyzing the problem in the light of Chomsky's Hierarchy of Formal Languages (see figure 6-1), this research identifies an appropriate theoretical model for developing a solution model. By determining requirements and identifying the most restrictive computational theory model that sufficiently expresses the problem, the solution avoids being overly complex. The computational theory model selected is used as the basis for creating a solution model.

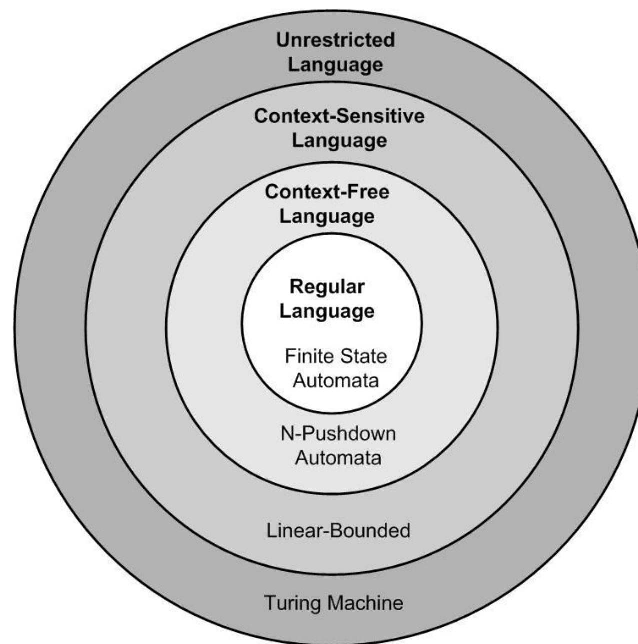

Figure 6-1 Chomsky's Hierarchy of Formal Languages

6.1 Criteria for a Computational Model

Bell and LaPadula claim it is important to "bridge the gap between general system theory and practical problem solving [Bell73]." Their research emphasizes the necessity of using computational models to guide solutions for IT security issues. A mathematical model allows researchers to represent system requirements and rigorously analyze them. Existing research on Intrusion Detection uses computational theory models associated with regular or context free

المنارة للاستشارات
www.manaraa.com

languages, but does not provide justification with respect to the selection of one theoretical model over another, nor does it exploit the theoretical characteristics of the selected model [Ko94] [Kos97] [Warr99]. This research explains its choice for a computational theory model, and shows how the theoretical characteristics of the model can be used to enforce a policy permitting only *valid process invocations sequences* (*VPIS*). The intent of analyzing computational models to find an appropriate theoretical framework is to preclude potential solution-related issues that might not otherwise present themselves prior to implementation of the solution.

6.1.1 . Necessary & Sufficient

Traditional logic's criteria of necessity and sufficiency are used to assist in the selection of an appropriate computational model to use in this research. If, when some condition occurs, an event associated with that condition also occurs, the condition is said to be sufficient for that event. However, the existence of an event does not imply that a specific sufficient condition has occurred since some sufficient conditions may be replaced by other conditions that are also sufficient. On the other hand, if a condition is necessary for the occurrence of an event, the occurrence of the event implies that the condition must have occurred. In other words, the event occurs if and only if the necessary condition occurs [Bark80] [Copi78].

The computational model selected in this research is sufficient to model the computational events investigated here. Turing Machines (TM) and their computational equivalents are sufficient to model any computational event. But TMs are not necessary to model all computational events. This research finds the computational model with the least computational power that still provides the computational power required to model the event.

المنارة للاستشارات
www.manaraa.com

In an argument, the set of necessary conditions contain only the essential conditions that support the conclusion. All conditions that are not necessary to support the conclusion may be removed to simplify the solution. Unnecessary premises can obscure an argument and make it difficult to follow [Copi78]. Similarly, in designing engineering solutions, unnecessary requirements can cause an engineer to loose focus on solving the problem. The logical principle of Ockham's razor suggests that the simpler solution is usually the better one. Although more complex solutions may also solve a problem, a better solution involves only those conditions that are necessary to realize the solution. Ockham's razor can be used general guide in the selection of the theoretical computational model to avoid unnecessary complexity [Gau03]; this approach is taken in this research.

6.1.2 Choosing a Computational Model

To identify the simplest computational model necessary to model the problem of validating a process invocation sequence instances of invocation sequences are represented in symbolic form. After representing the instances in symbolic form, the validation problem is evaluated using computational models from the simplest to the most complex, as required, stopping at the simplest computational model necessary to model the event. By evaluating the simplest computational models first, and only moving on to more complex models if it is determined that a simpler model is inadequate, this research can be certain that it bases its solution on the simplest necessary computational model.

The computational model associated with the theory chosen should have the capacity to accept or reject a candidate string representing a process invocation sequence. The computational model must have the ability to recognize the set of strings in a language $\boldsymbol{L}$ in which each string represents a finite sequential invocation of processes during execution of an

المنارة للاستشارات
www.manaraa.com

operating system so long as each process is invoked by a process that is authorized to invoke it. Any string representing some other process invocation sequence should be rejected as not belonging to language ***L***.

6.1.2.1 Multitasking Requirement

Within the computational model it is necessary to represent an executing process and the sequence of process invocations. This is complicated by the fact that multiple invocation sequences can execute with apparent parallelism. This research makes the simplifying assumption that a system contains a single CPU. The ability of computers to run multiple unrelated process sequences, which appear to execute simultaneously, is called multitasking. A user's series of processes, although logically executed in a sequential chain, may not actually be scheduled in an uninterrupted, temporally contiguous sequence by the operating system's scheduler. Rather, the scheduler may find it necessary to interrupt that process invocation sequence by starting or resuming one or more other process invocation sequences needing to run on the single CPU. As a result, a number of unrelated process invocation sequences may be competing for time on a single CPU processor. Or, a server may process multiple functions within an application on behalf of multiple users, creating multiple process invocation sequences. Multitasking presents a challenge requiring a computational model capable of evaluating multiple simultaneous process invocations sequences.

Take two jobs submitted for execution to a server, job ***a*** is represented by the process invocation sequence ***abcabc*** and job ***x*** is represented by the process invocation sequence ***xwybzw***. Assume that job ***a***: ***abcabc*** and job ***x***: ***xwybzw*** each represent *valid process invocations sequences* with each symbol in a string corresponding to the process being invoked. Also assume that a sequence of processes associated with job ***a*** is executing on behalf of user 1 and a

المنارة للاستشارات
www.manaraa.com

sequence of processes associated with job $\boldsymbol{x}$ is executing on behalf of user 2. To the scheduler and CPU, however, the execution of the processes associated with the two strings may actually occur as shown in figure 6-2:

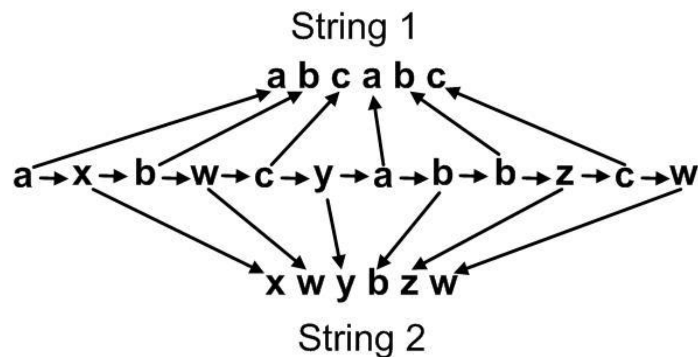

Figure 6-2: Interleaved Process Execution

This happens because, when a process invokes another process, an interrupt occurs, allowing the scheduler to dispatch another process sequence or system service. System interrupts cause interleaving of process sequences, meaning, effectively, that multiple process invocation sequences must be simultaneously evaluated for authorization.

6.1.2.2 Regular Language

From the perspective of formal language theory, a Finite State Automata (FSA) is the simplest computational machine. Other researchers have used a FSA in an attempt to map *valid process invocations sequences* in different forms. In intrusion detection research, which also attempts to map *valid process invocations sequence*, both Kosoresow [Kos97] and Sekar [Sek01] entertained the use of the FSA approach.

Kosoresow identified a Deterministic Finite State Automata (DFA) as an appropriate approach to identify patterns with in an invocation sequence [Kos97]. However, Kosoresow also identified some issues using a DFA. For instance, (1) using a DFA to calculate the minimal description of a *valid process invocation sequence* may be time consuming and is potentially NP-

المنارة للاستشارات
www.manaraa.com

hard [Kos97]. (2) Creating a DFA from valid process invocations sequence has do be done heuristically, using scripts and manually. Therefore, an exact DFA representation is likely to be problematic. And (3) creating a DFA from traces would require substantial space (memory) [Kos97]. A description of Kosoresow's approach can be found in section 5.2.4.1.2.

Sekar, on the other hand, embraced the Finite State Automata (FSA) as an approach to solve the problem of verifying *valid process invocation sequences*. Sekar's approach solved the problem of manual learning [Sek01], although it can be argued that Sekar did not address the issues of the computational expense and the excessive space needed to implement this model as raised by Kosoresow [Kos97]. By way of contrast, the present research uses the concept of "one state, one process" in its implementation to solve these issues (see chapter 7). A description of Sekar's approach can be found in Section 5.2.4.1.3.

Other, more complex, conceptual models have also been used, such as Push Down Automata (PDA) [Feng03] and the probabilistic Hidden Markov Model (HMM) [Gho00]. For more information on other research using a FSM and other theoretical computational methods to map *valid process invocation sequences*, see section 5.2.4. But recall that a more complex solution, although sufficient, may not be logically necessary.

A FSA recognizes a regular language. Therefore, this model is appropriate if the set of strings representing process invocation sequences form a regular language. Each language accepted by some FSA has a corresponding representation in a Regular Expression (RE). A RE is defined [Sips06] [Rich08] as follows:

المنارة للاستشارات
www.manaraa.com

Regular Expression Definition

String S is a **regular expression** over the alphabet Σ, if S is:

1. a for some a in alphabet Σ
2. ε
3. $\varnothing$
4. (S) is a regular expression
5. Union: $(S_1 \cup S_2)$, where S_1 and S_2 are regular expressions.
6. Concatenation: $(S_1 \circ S_2)$, where S_1 and S_2 are regular expressions.
7. Kleene Star: S^*
8. Kleene Plus: S^+

Figure 6-3: Definition of Regular Expression

A language L is regular if the set of strings in L corresponds to some regular expression string as defined by the definitions above. For example, if strings corresponding to regular expression S_1 are in some regular language L_1 and strings corresponding to regular expression S_2 are in the regular language L_2, then a regular expression can be formed corresponding to the language consisting of exactly the union of the set of strings corresponding to S_1 with the set of strings corresponding to S_2. Let one set of strings in the language corresponding to S_1 represent a set of *valid process invocation sequences* and let another set of strings corresponding to S_2 represent another set of *valid process invocation sequences.* Both sets are part of a regular language described by the union of their respective corresponding regular expressions. Let one FSA be constructed to recognize S_1, and the other S_2. Then, an FSA can be constructed to recognize the sets of strings corresponding to both S_1 and S_2, using ε transitions from a new start state to the original start states of the two FSAs previously used to recognize each set individually. The resulting non-deterministic FSA now recognizes a new set of *valid process invocations sequences* containing all the members of S_1 and all the members of S_2.

المنارة للاستشارات
www.manaraa.com

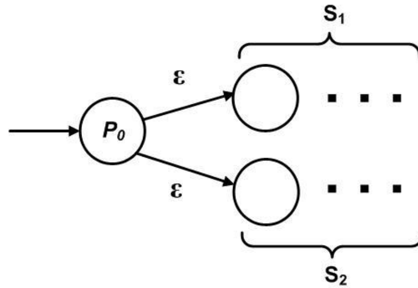

Figure 6-4: Representation of the FSA Recognizing the Union of Languages S1 and S2

Figure 6-4 illustrates the fact that Regular Languages are closed under union.

6.2 Appropriate Representation of the Problem

Any two FSAs (whether deterministic or non-deterministic) are said to be equivalent if they recognize exactly the same language [Hopc01] [Rich08] [Sips06]. Furthermore, it is known that every non-deterministic FSA (NFA) has an equivalent deterministic FSA (DFA). Every NFA can be algorithmically transformed into its equivalent DFA [Brzo62] [Hopc01] [Rich08] [Sips06] [Wats95] [Wats00]. The PPT models the process invocation sequence problem using a DFA.

6.2.1 Finite State Automata Representation

A 5-tuple definition of a FSA ($\boldsymbol{Q}$, Σ, δ, $\boldsymbol{P_0}$, $\boldsymbol{F}$) as shown in figure 6-5 (modified from [Sips06]) is the starting point for defining the elements required to represent a process invocation sequence. As detailed below, the DFA is used to accept or reject a process invocations, represented as a string over $\boldsymbol{\Sigma}$, with the states in $\boldsymbol{Q}$ representing the processes, i.e. $\boldsymbol{Q}$ = **{*processes*}**. This same representation could be used even if a more computationally powerful machine were required. That is: the processes are represented by the states in $\boldsymbol{Q}$, and the invocation sequences are represented by a sequence of symbols over $\boldsymbol{\Sigma}$. Therefore both $\boldsymbol{Q}$ and $\boldsymbol{\Sigma}$ are associated with processes, where $\boldsymbol{Q}$ = **{*processes*}** and $\boldsymbol{\Sigma}$ = **{*process invocations*}**.

المنارة للاستشارات
www.manaraa.com

Deterministic Finite State Automata Definition
$(Q, \Sigma, \delta, P_0, F)$

Q is a finite set of states.

Σ is a finite alphabet.

$\delta : Q \times \Sigma \rightarrow (Q\text{-}\{P_0\})$ is the transition function.

$P_0 \in Q$ is the unique start state.

$F \subseteq Q$ is the set of accepting states.

Figure 6-5: Definition of Deterministic Finite State Automata

Both nondeterministic FSAs (NFA) and deterministic FSAs (DFA) are used throughout this chapter to describe the program pathing solution. Because an algorithm exists to convert an arbitrary NFA to an equivalent DFA [Sips06], either one is used as convenient. The primary difference between the deterministic and nondeterministic definition is step 3 in figure 6-5, where the transition function δ of the deterministic FSA is replaced with the transition relation Δ of the nondeterministic FSA, where $\Delta: Q \times (\Sigma \cup \{\varepsilon\}) \times (Q\text{-}\{P_0\})$. [Rich08]

6.2.1.1 States Q

In the finite set Q, P_i represents process i executing in the system. More formally, $Q = (\{P_0\} \cup \{P_i : P_i$ represents a process P_i that has been dispatched to execute by the scheduler$\})$. States are labeled as $P_0, P_1, P_2, \ldots, P_n$, where the symbol P_n represents the name of a real process. $|Q| = n+1$, there n is determined by the number of processes represented in Q. If there are five processes, then there are five states in the DFA, plus one for the start state P_0. All states in Q represent processes that are authorized to execute on the computer. P_0 is a start state (described in section 6.2.1.4) and is not reachable after transition to another state in Q.

المنارة للاستشارات
www.manaraa.com

6.2.1.2 Alphabet Σ and process invocations

Normal processes are those processes that are allowed to execute on the critical server as defined in chapter 2 to fulfill its critical function. $\boldsymbol{\Sigma}$ is the alphabet of all the transition inputs of normal process invocations corresponding to the set of process represented in $\boldsymbol{Q}$ and are represented as lower case $\boldsymbol{p}$. Every process, whether normal or not, that can be invoked on a system is associated with a corresponding symbol in a finite, though possibly large, alphabet $\boldsymbol{\Sigma}$. Symbol $\boldsymbol{p_i \in \Sigma}$ represents the invocation of process $\boldsymbol{P_i}$. These symbols are used to form strings representing all possible process invocation sequences in a language $\boldsymbol{L}$. The alphabet is a finite set of symbols, therefore, over which all possible process invocation sequences both *valid* and *invalid* are formed. Thus while $\boldsymbol{P_i \in Q\text{-}\{P_0\}}$ represents a process named $\boldsymbol{P_i}$ executing, $\boldsymbol{p_i \in \Sigma}$ represents the invocation of process $\boldsymbol{P_i}$. The language constructed from $\boldsymbol{\Sigma}^*$ is the language consisting of the set of all possible strings that can form over the alphabet $\boldsymbol{\Sigma}$, that is, $\boldsymbol{\Sigma} =$ $\{\boldsymbol{p_1, p_2, p_3, \dots, p_n}\}$, where $|\boldsymbol{\Sigma}| = \boldsymbol{n}$.

When process $\boldsymbol{P_i}$ invokes $\boldsymbol{P_k}$, it causes a transition in the NFA represented as $\boldsymbol{\Delta(\{P_i\},p_k)}$. This transition represents that while executing the process $\boldsymbol{P_i}$, the system encounters the invocation of a process $\boldsymbol{P_k}$ as a transition input (or invocation) represented as $\boldsymbol{p_k}$. Because not all possible process invocations are valid process invocations, it is possible that $\boldsymbol{\Delta(\{P_i\},p_k)}$ is not defined for some $\boldsymbol{p_k \in \Sigma}$ or for some $\boldsymbol{P_i \in Q}$. For example, some processes should never be allowed to execute on a critical server, because these processes are not essential to the primary function of the server. These *invalid process invocations* are represented by a subset alphabet $\boldsymbol{\Sigma_\beta \subseteq \Sigma}$. The set of *valid process invocations* in $\boldsymbol{\Sigma}$ is represented by the subset alphabet $\boldsymbol{\Sigma_\gamma}$, where $\boldsymbol{\Sigma_\gamma \subseteq \Sigma}$, with $\boldsymbol{\Sigma}$ being partitioned by $\boldsymbol{\Sigma_\beta}$ and $\boldsymbol{\Sigma_\gamma}$. That is:

$$\boldsymbol{\Sigma_\gamma \cup \Sigma_\beta = \Sigma}$$

المنارة للاستشارات
www.manaraa.com

and

$$\Sigma_\gamma \cap \Sigma_\beta = \emptyset$$

The set of *invalid process sequences* is of interest. Any process invocation sequence that contains a non-empty substring from the alphabet Σ_β by definition is an *invalid process invocation*. It is invalid because it contains at least one symbol corresponding to the invocation of an *abnormal process*. Since Σ is a set of *process invocations*, there are two possible subsets for any sequence in the Σ set of *invalid process invocations sequences*.

- $\{w \in (\Sigma_\beta \cup \Sigma_\gamma)^+$, where w contains at least one symbol from the alphabet Σ_β.}, which represents the set of *invalid process invocations* made up of *normal* and *abnormal processes*. Any string made up of a symbol not belonging exclusively to the alphabet $\Sigma_\gamma$is considered invalid.
- Σ_β^+ represents the set of *invalid process invocations* made up of only *abnormal processes*. Any string made up of a symbol not belonging to the alphabet $\Sigma_\gamma$is considered invalid.

$$\Sigma_\beta^+ \subseteq w$$

$$\text{where,} \qquad w \in (\Sigma_\beta \cup \Sigma_\gamma)^+$$

The set Σ_γ^* contains all the possible strings that can be created from the alphabet Σ_γ. Thus, Σ_γ^* contains all possible strings representing *valid process invocation sequences* over a language using Σ_γ as the alphabet, and more generally, Σ. Σ_γ^* may also contain some *invalid process invocation sequences*. This is because for a process invocation to be valid, it is required that the order of its symbols appear in an appropriate sequence and that the sequence be profiled. It is possible that an *invalid invocation process sequence* could be constructed from the set of valid process invocations. An *invalid process invocation sequences* over Σ_γ represents the case in

المنارة للاستشارات
www.manaraa.com

which *normal processes* are invoked in a sequence that is not appropriate. The language L_x is the language made up of the subset of Σ_γ^* that makes up the set of *invalid process invocation sequences*. The set of strings Σ_γ^* containing the subset of *invalid process invocation sequences* L_x such that:

$$L_x \subset \Sigma_\gamma^*$$

The strings in the language L_x represent process invocation sequences made up of *valid process invocations* from the alphabet Σ_γ, but there is at least one substring w in each string such that $|w| \geq 2$ and w represents an *invalid process invocation*.

Consider a system process invocation sequence: $P_q P_r P_s P_t$. A *normal process* can only execute if it is invoked by another *normal process* and if that invocation is valid. Thus P_s executes, because it has been invoked by process P_r. Similarly, process P_t executes because it was invoked by process P_s. The ordinal execution of these processes is represented as a sequence over Σ by the following, $p_r p_s p_t$. As the OS schedules a finite sequence of processes for execution, a corresponding finite-length sequence of symbols w over Σ_γ is formed. At any invocation index t in the scheduled sequence, the length of w at invocation index t is represented as $|w_t|$. When the next process is scheduled, a new string w_{t+1} is created such that $|w_{t+1}| = (|w_t|+1)$.

The set of *valid process invocations sequences* is the language L_v, over the alphabet Σ_γ. In L_v each symbol in a string represents invocation of a *normal process* that is authorized to invoke the subsequent process represented by the subsequent symbol in that string. The set of *valid process invocation sequences* is a possibly infinite set of such finite-length strings forming the language L_v. Recognition of the language L_v is the focus of this research.

www.manaraa.com

$\boldsymbol{\Sigma}_{\gamma}^{*}$ can be divided therefore, into two distinct partitions, $\boldsymbol{L_x}$ and $\boldsymbol{L_v}$ where:

$$\boldsymbol{L_v} \subseteq \boldsymbol{\Sigma}_{\gamma}^{*}$$

$$\boldsymbol{L_x} \subseteq \boldsymbol{\Sigma}_{\gamma}^{*}$$

$$\boldsymbol{\Sigma}_{\gamma}^{*} = \boldsymbol{L_x} \cup \boldsymbol{L_v}$$

$$\boldsymbol{L_x} \cap \boldsymbol{L_v} = \emptyset$$

That is: the set of *invalid process invocation sequences* $\boldsymbol{L_x} = (\boldsymbol{\Sigma}_{\gamma}^{*} - \boldsymbol{L_v})$, represents the set of process invocation sequences that are invoked in an *invalid* order, although made up of only symbols representing *valid process invocations*, while the set of possible *valid process invocation sequences* $\boldsymbol{L_v} = (\boldsymbol{\Sigma}_{\gamma}^{*} - \boldsymbol{L_x})$, represents the set of *valid process invocation sequences*.

6.2.1.3 Transition Relation Δ

An invocation of process $\boldsymbol{P_2}$ by process $\boldsymbol{P_1}$ is expressed in the NFA transition relation as $\Delta(\{\boldsymbol{P_1}\},\boldsymbol{p_2}) = \{\boldsymbol{P_2}\}$. This representation indicates a transition from the NFA set of states containing $\boldsymbol{P_1}$ to the set of states containing $\boldsymbol{P_2}$ on the input symbol $\boldsymbol{p_2}$. The inferred substring $\boldsymbol{w}$ is represented as the sequence $\boldsymbol{w} = (\boldsymbol{p_1 p_2})$ over $\boldsymbol{\Sigma_v}$. Let $\boldsymbol{p_j}$ represent invocation of process $\boldsymbol{P_j}$ immediately subsequent to $\boldsymbol{w}$. Then a new process invocation sequence is formed by the concatenation $\boldsymbol{wp_j}$.

At invocation index $\boldsymbol{t_0}$ no process has been invoked. $\boldsymbol{P_0}$ is an accepting state. Therefore the initial language is defined as $\boldsymbol{L_{t0}} = \{\boldsymbol{\varepsilon}\}$. The NFA recognizing $\boldsymbol{L_{t0}}$ is called $\boldsymbol{NFA_{t0}} = (\boldsymbol{Q}=\{\boldsymbol{P_0}\},\boldsymbol{\Sigma}, \boldsymbol{\Delta}, \boldsymbol{P_0}, \boldsymbol{F}=\{\boldsymbol{P_0}\})$ as shown graphically in figure 6-6.

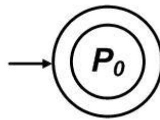

Figure 6-6 Initial Start State – NFA_{t0}

المنارة للاستشارات
www.manaraa.com

Assume for the moment that every process invocation would occur in a valid sequence. Then for invocation of process $\boldsymbol{P_1}$ we consider $\boldsymbol{NFA_{t1}}$ formed by replicating $\boldsymbol{NFA_{t0}}$ and adding a transition on input $\boldsymbol{p_1}$ to a new accepting state $\boldsymbol{P_1}$. As a result, $\boldsymbol{NFA_{t1} = (Q=\{P_0,P_1\},\Sigma=\{p_1\}, P_0, \Delta, F=\{P_0,P_1\})}$ as shown in graphically in figure 6-7 recognizes $\boldsymbol{L_{t1}= \{\varepsilon, p_1\}}$.

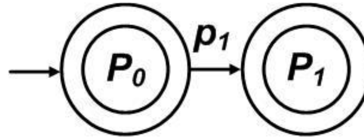

Figure 6-7 NFA$_{t1}$

As each invocation progresses and adds another process invocation to the string, a new language is created consisting of the previous language and adding a new string. Assume string $\boldsymbol{p_1}$ has resulted in the machine $\boldsymbol{NFA_{t1}}$ being in the state $\boldsymbol{P_1}$ at invocation index $\boldsymbol{t_1}$. Then at index $\boldsymbol{t_2}$ with next process invocation $\boldsymbol{p_2}$, a transition resulting in sting $\boldsymbol{p_1p_2}$ is recognized by machine $\boldsymbol{NFA_{t2} = (Q=\{P_0,P_1,P_2\},\Sigma=\{p_1,p_2\},\Delta, P_0, F=\{P_0,P_1,P_2\})}$ which recognizes the language $\boldsymbol{L_{t2} = \{\varepsilon, p_1, p_1p_2\}}$.

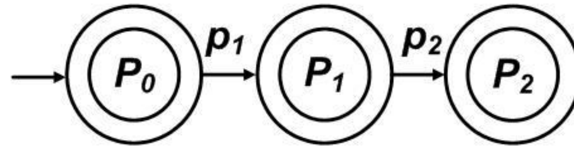

Figure 6-8 NFA$_{t2}$

Similarly, invocation index $\boldsymbol{t_3}$, where the transition relation $\boldsymbol{\Delta(\{P_2\},p_3) = \{P_3\}}$, the transition represents the addition of $\boldsymbol{p_3}$ process invocation. Machine $\boldsymbol{NFA_{t2}}$ adds the string $\boldsymbol{(p_1p_2p_3)}$, which recognizes a new language $\boldsymbol{L_{t3} = \{\varepsilon, p_1, p_1p_2, p_1p_2p_3\}}$ resulting in the machine $\boldsymbol{NFA_{t3} = (Q=\{P_0,P_1,P_2,P_3\},\Sigma=\{p_1,p_2,p_3\},\Delta, P_0, F=\{P_0,P_1,P_2,P_3\})}$ The sequence of invocation event indices occurs until the process sequence terminates and the all process invocation sequences are mapped.

المنارة للاستشارات
www.manaraa.com

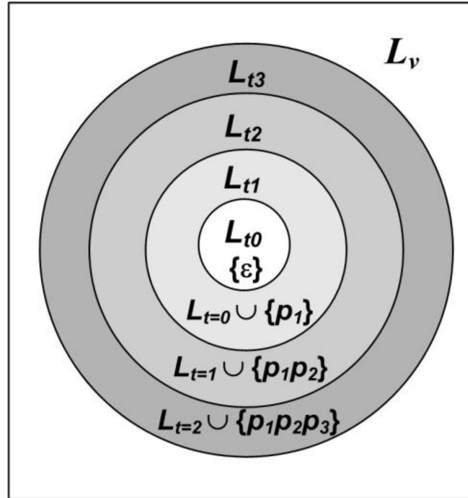

Figure 6-9 Building the Valid Process Invocation language

As each new *normal process invocation* is concatenated to some string present in the current set of *valid process invocation sequences*, a new language is created. The building of the sets of process invocation sequences is symbolized:

$$L_v \subseteq \Sigma_\gamma^*$$

Initially, $L_{t0} = \{\,\}$

$$L_{t0} \subseteq L_v$$

because, $L_{t1} \subset L_{t2} \subset L_{t3} \subset \Sigma_\gamma^*$,

we know that $|\Sigma_\gamma^*| > |L_{t3}| > |L_{t2}| > |L_{t1}| > |L_{t0}|$

Figure 6-9 illustrates the building of the new language as it grows in graphic form.

Note that the series of machines constructed in this manner do not recognize Σ_γ^*, but only increasingly large subsets of that language. *Invocation index based string construction* allows for validating a process invocation sequence as the sequence is processed by the system's scheduler.

المنارة للاستشارات
www.manaraa.com

6.2.1.4 Start State P_0

An assumption is that only process invocations that occur subsequent to a machine being in state $\boldsymbol{P_0}$ are of interest. In a computing system this can be any assumed environment. It can be a user pushing a power start button, issuing a command, clicking on a GUI. Or it can be the state after a computer's OS has been booted. In this research the symbol $\boldsymbol{P_0}$ is used to symbolize any assumed safe starting condition. $\boldsymbol{P_0}$ is not an executing process.

6.2.1.5 FSA Issues

This research represents a series of process invocations as an FSA. The standard 5-tuple definition of an FSA can be used with the definitions above to model process invocation sequences. Let a machine PPTM (Program Pathing Trust Machine) be an FSA with process states $\boldsymbol{Q = \{P_0, P_1, P_2, P_3, \ldots, P_n\}}$. $\boldsymbol{P_0}$ is not an executable process and $\boldsymbol{P_1}$ through $\boldsymbol{P_n}$ are executable processes. The alphabet $\boldsymbol{\Sigma}$ corresponds to the set of process invocations (see section 6.2.1.2) $\{\boldsymbol{p_1, p_1, p_3, \ldots p_n}\}$. $\boldsymbol{P_0}$ represents the initial start state of the PPTM. The set of final or accepting states is represented by $\boldsymbol{F \subseteq Q}$. Process invocation transitions are defined as relation.

$$\boldsymbol{\Delta: Q \times (\Sigma \cup \{\varepsilon\}) \times (Q - \{P_0\})}$$

Mapping process invocation sequences (strings) over an alphabet $\boldsymbol{\Sigma_\gamma}$ is clear enough. However, this mapping does not resolve the issue of which computational model is needed to solve the problem of determining *valid invocation sequences*. Validating the authority of a *normal process* to invoke another *normal process* is also a requirement and does not just entail a strict mapping of each instance of a process invoked by the scheduler. This is a mistake made by some other solutions proposed to solve the problem [Amm98] [Ball96]. However this research makes the assumption that once a process is authorized to invoke another process, it is valid for

المنارة للاستشارات
www.manaraa.com

the invocation to be repeated, no matter how many times. This assumption is explained in this section and in section 6.2.2.

Suppose ($\boldsymbol{p_1 p_2 p_3 p_3 p_4}$) represents a *valid process invocation sequence*. For the program pathing problem, the regular expression representing this process invocation sequence is ($\boldsymbol{p_1 p_2 p_3^+ p_4}$). The regular expression does not count the number of times $\boldsymbol{p_3}$ occurs in the sequence, but does indicate that $\boldsymbol{p_3}$ must occur in the sequence, at least once, subsequent to $\boldsymbol{p_2}$ and prior to $\boldsymbol{p_4}$. Since the regular expression does not have to insure that a sequence be exact in the number of times a process be invoked, the language is regular. The above regular expression specifies that $\boldsymbol{p_1}$ precedes $\boldsymbol{p_2}$, $\boldsymbol{p_2}$ precedes $\boldsymbol{p_3}$, and that $\boldsymbol{p_3}$ can precede itself and precedes $\boldsymbol{p_4}$. Once it is established that $\boldsymbol{P_3}$ may validly invoke $\boldsymbol{P_3}$ then this invocation can occur each time the machine is in $\boldsymbol{P_2}$ and can occur any number of times.

Validating the authority of a process to invoke another process is not dependent upon the number of times the process is invoked. Proving the correctness of execution of some portion of the computation may require counting invocations, but that is a different problem. The program pathing problem requires validating that a process has the authority to invoke another process and is different from verifying the correctness of the execution. A process may invoke or fail to invoke a *normal process* that it is authorized to invoke, yet the process is still authorized to perform the invocation. Whether or not the process performs the correct invocation may involve a program logic error, but is not a question of validating the process's authority to perform the invocation. For instance, a teleprocessing program may invoke a process three times to serve three different users, but if only two of the processes invoked the ending process and the third process ends in error, then the third ending process is never invoked. This use case would constitute a process flow logic error, but not an *invalid invocation sequence*. A language

المنارة للاستشارات
www.manaraa.com

containing strings that need to represent how many times a *normal process* has been invoked would not be a regular language and would require a more complex machine to recognize the language. If a more complex language were required, a context free language would have to be considered.

6.2.2 Regular vs. Context Free Language

Without the need for a language to pump a symbol of the alphabet a specific number of times, it is not necessary to use a machine more powerful than an finite state automata. The classic example of a language that does not meet the criteria for the regular language pumping lemma is $\boldsymbol{L} = \{\boldsymbol{p_1^n p_2^n}\} |\ \boldsymbol{n} \geq \boldsymbol{0}\}$. For $\boldsymbol{L}$ it is necessary to count the number of times a process is called, assuming $\boldsymbol{L}$ is a set of strings representing correct process invocation sequences. However, the program pathing problem does not involve verifying that the process invocation sequence is logically correct, but that the process invocation sequence is authorized. Anytime a symbol occurs consecutively, it can be replaced in the corresponding regular expression by the Kleene plus. This means that a string of the form $\boldsymbol{A^n B^n}$ for $\boldsymbol{n>0}$ in the context of this problem has the equivalent regular expression $\boldsymbol{A^+ B^+}$. As a result the language is no more complex than a regular language [Bar61] [Hopc01] [Rich08] [Sips06].

6.3 Finite State Automata and the Program Path Trust Model

Given that an FSA is a reasonable theoretical foundation upon which to build the Program Pathing Trust Machine (PPTM) and that it is possible to represent that problem symbolically as described earlier, it is necessary to show how the PPTM can be constructed. Let $\boldsymbol{M}$ be a computational model (machine). Let $\boldsymbol{L(M)}$ be the language recognized by machine $\boldsymbol{M}$.

المنارة للاستشارات
www.manaraa.com

Let L_v (M) represent the language recognized by some machine M and containing all the strings representing the *valid process invocation sequences*. The symbolic representation for this machine has been discussed in section 6.2.

A sequence of process invocations is defined as an ordered series observed prior to some fixed event at invocation index t as a computing system schedules consecutive processes. The set of all possible invocation sequences forms the set Σ^*. This language may be viewed as two subsets, as described in section 6.2. One subset L_v (M) contains exactly the set of sequences corresponding to the series of acceptable process invocations of the system. The other subset $\overline{L_v(M)}$ contains the set of sequences corresponding to the series of *invalid process invocations* in the system that are known to be invalid. The two subsets are disjoint and partition the set of all possible process invocation sequences (see section 6.2.1.1).

Taking both subsets into account, we can create a corresponding DFA. The FSA solution model proposed in this research initially assumes that the set of *valid invocation sequences* is $\{\varepsilon\}$. As the solution model identifies and adds new *valid process invocations sequences* to the set, the set of *valid invocation sequences* grows. The solution model ensures system integrity by identifying *valid process invocation sequences*. Techniques for distinguishing these subsets are discussed in detail in sections 6.3.2 and 6.3.3. Over time, by building the set $L \subseteq L_v$ (M), we can reduce the occurrences of rejection of *valid process invocation sequences*, and allow the system to identify *valid sequences* that it may not have explicitly recorded. However, this means there may be some process invocation sequences that are inferred by M_t, but have not been encountered. These process invocations sequences may or may not be valid, but cannot be determined, without application of domain knowledge (as discussed in section 6.3.3 and following).

www.manaraa.com

6.3.1 Why Finite State Automata is a better Computational Model choice.

One possible concern with the FSA approach is that in a real system many processes are invoked and the FSA might quickly become very large and complex, especially if it is to be deterministic. This concern can be addressed to some extent by using minimization algorithms for DFAs as described by Hopcroft [Hopc01] and Watson [Wats95], or by the minimization algorithm for non-deterministic FSAs described by Brzozowki [Brzo62]. Application of such minimization algorithms can yield a simpler equivalent machine with fewer states.

Another concern with the use of FSA, from the machine learning perspective, is that while learning by rote, it may appear that the FSA cannot abstract or generalize from the data that it profiles. However, the process of generating equivalent states through application of the DFA minimization algorithm can be viewed as generalization. This generalization is accomplished in two ways. (1) The looping structures allow for the machine to recognize process invocation sequences it has not encountered previously, such as multiple invocations of a process or set of processes. (2) The FSA can encounter multiple prefix invocation subsequences leading to some configuration from which machine $\boldsymbol{M}$ transitions to an accepting state, using the remaining portion of another input process invocation subsequence to represent a novel process invocation sequence.

6.3.2 Finite State Automaton PPT Representation

The set of *abnormal processes* is a finite set of processes that should never be permitted to be invoked in the computer. This set is symbolized as $\boldsymbol{\mu}$, as the set of abnormal process invocations.

$$\boldsymbol{\mu} = \boldsymbol{\Sigma}_{\beta} \cup \boldsymbol{G}$$

$$\text{where, } \boldsymbol{G} \subseteq \boldsymbol{\Sigma}_{\gamma}$$

المنارة للاستشارات
www.manaraa.com

The set $\boldsymbol{\mu}$ is the set of partially indeterminate processes. It includes the set of *abnormal processes* from the alphabet Σ_β. However, it also includes the set $\boldsymbol{G}$, which is the set of all *normal process* from the alphabet Σ_γ whose transitions and states have not yet been represented in the DFA. The set $\boldsymbol{G}$ is important, because it represents those *normal processes* not considered valid to be invoked some time. Once a *normal process* from $\boldsymbol{G}$ is considered authorized to be invoked, it is removed from the set $\boldsymbol{G}$, and therefore removed from $\boldsymbol{\mu}$.

Valid process invocation sequences (*VPIS)* are the only process invocation sequences that can execute. The system runs continuously as long as the process invocation sequences are valid. When a *valid process invocation* is encountered, the DFA transitions to an accepting state. When an *invalid process invocation* is encountered, the DFA enters a trap state from which it cannot escape. Since this state is not an accepting state, the DFA can not recognize any process invocation sequence that causes it to enter the trap state, and therefore the sequence is determined to be invalid. The $\boldsymbol{DFA_t}$, recognizes the subset of the language $\boldsymbol{L_v}$, i.e., it recognizes a subset of *valid process invocation sequences.*

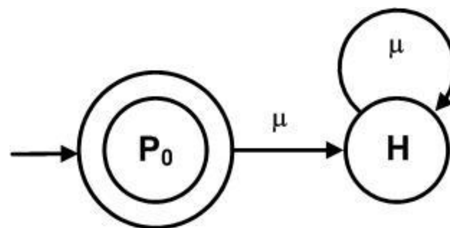

Figure 6-10 $DFA_{t=0}$ **Transition Diagram**

$\boldsymbol{DFA_{t=0}}$ (figure 6-10) initially accepts the empty string, $\boldsymbol{\varepsilon}$, and rejects *invalid process invocation sequences* and $\boldsymbol{L(DFA_{t=0})}$. In this initial machine no actual process invocation sequence is valid. Any process invocation sequence that is presented to the $\boldsymbol{DFA_{t=0}}$ is invalid, and is not recognized by the DFA. As processes are encountered, they are validated against the DFA's states, and are assumed valid as long the input invocation transition causes a transition to

www.manaraa.com

an accepting state. If the input invocation causes a transition to ***H*** (the trap state), then the process invocation sequence is determined to be invalid.

6.3.3 PPT Finite-state Automata Learning Mode

It has been established that a DFA can be used to represent a process invocation sequence. However, the challenge is in populating the PPT DFA with *valid process invocation sequence*s. The most straight forward approach is to initially train the DFA with valid training data. The process invocations the DFA profiles by rote represent *valid process invocation sequences* that can be audited by a person.

The method used in this research is to provide a profiling mode. An administrator can turn on profiling while accessing a particular application. When the profiling mode is turned on, the sequence of application process invocations are introduced to the DFA as training data. New strings in the language L_v are recognized by the DFA as they are encountered. As a result a new DFA is created as necessary with appropriate transitions with all other input transitions to the new state being set to the default of transitioning to the trap state (see figure 6-11). Therefore, as each *normal process* is validly invoked, transitions are added to the DFA until the entire *valid process invocation sequence* is recognized. Consider a newly initialized PPT DFA before it encounters any *normal process invocations*. Its transition table is shown in figure 6-11; here and afterwards, the start state appears in the first row of the transition table and all accepting states are shown in boldface and underlined type.

Input

State		μ
	$\underline{P_0}$	H
	H	H

Figure 6-11: Initial PPT DFA Translation Table

Consider, a $DFA_{t=1}$ where the *valid process invocation sequence* ($p_1\, p_1\, p_3$) is encountered.

المنارة للاستشارات
www.manaraa.com

Remember,

$$L(DFA_{t=0}) = \{\varepsilon\}$$

and, $L(DFA_{t=0}) \subseteq L_v$

As the $DFA_{t=0}$ encounters p_1, symbol p_1 representing a previously unseen process is encountered, that symbol is removed from μ creating the set μ - $\{p_1\}$. The encounter of the first process invocation p_1 in the sequence ($p_1p_1p_3$) causes a new DFA machine to be created:

$$L(DFA_{t=1}) = L(DFA_{t=0}) \cup \{p_1\}$$

As the $DFA_{t=1}$ encounters p_1 again, the second process invocation in the sequence ($p_1p_1p_3$) another new DFA machine is created.

$$L(DFA_{t=2}) = L(DFA_{t=1}) \cup \{p_1p_1\}$$

Recall that the corresponding regular expression for the substring p_1p_1 is $\mathbf{p_1^+}$. As a result, the loop back transition is create on the P_1 state as shown in figure 6-12. As the $DFA_{t=2}$ encounters p_3, the third process invocation in the sequence ($p_1p_1p_3$) another new DFA machine is created, as shown in figure 6-12.

$$L(DFA_{t=3}) = L(DFA_{t=2}) \cup \{p_1p_1p_3\}$$

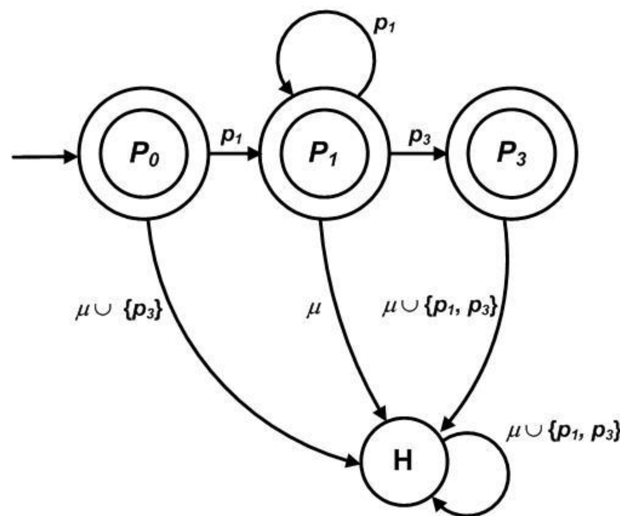

Figure 6-12 Transition Diagram Representing the DFA Recognizing Language $DFA_{t=3}$.

المنارة للاستشارات
www.manaraa.com

The DFA transition table for the machine $\boldsymbol{DFA_{t=3}}$ is shown in figure 6-13.

.	μ	p_1	p_3
$\underline{P_0}$	H	P_1	H
H	H	H	H
$\underline{P_1}$	H	P_1	P_3
$\underline{P_3}$	H	H	H

Figure 6-13 $DFA_{t=3}$ **Transition Table Learning**

As $\boldsymbol{DFA_{t=3}}$ is presented a new *valid process invocation sequence* $(\boldsymbol{p_1\, p_2\, p_3})$, and the first process invocation $\boldsymbol{p_1}$ is encountered, $\boldsymbol{p_1}$ is already accepted by $\boldsymbol{DFA_{t=3}}$, so no new DFA machine need be created. As $\boldsymbol{DFA_{t=3}}$ encounters the second process invocation, $\boldsymbol{p_2}$ in the sequence a new DFA machine is created.

$$L(DFA_{t=4}) = L(DFA_{t=3}) \cup \{p_1p_2\}$$

The result should be recognition of $\boldsymbol{L(DFA_{t=4})}$ adding the sequence $\boldsymbol{p_1\, p_2}$ to the language $\boldsymbol{L(DFA_{t=3})}$. As the $\boldsymbol{DFA_{t=4}}$ encounters $\boldsymbol{p_3}$, the third process invocation in the sequence $\boldsymbol{\{p_1p_2p_3\}}$ another new DFA machine is created.

$$L(DFA_{t=5}) = L(DFA_{t=4}) \cup \{p_1p_2p_3\}$$

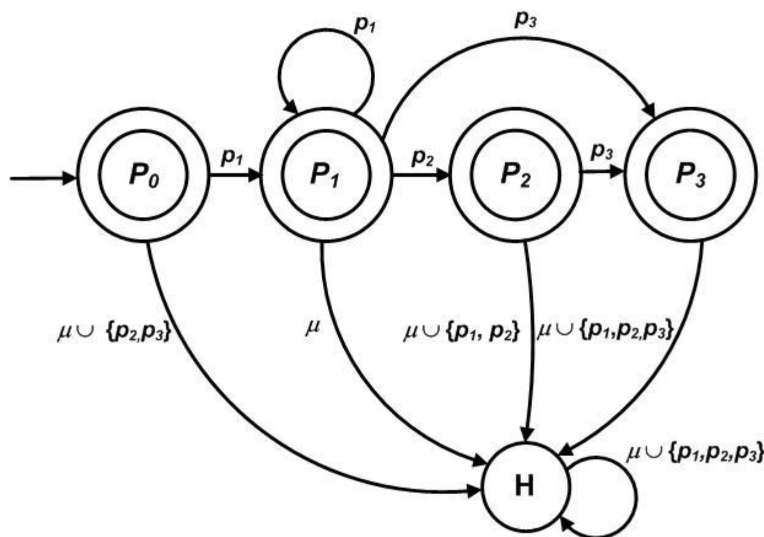

Figure 6-14 $DFA_{t=5}$ **Transition Diagram**

المنارة للاستشارات
www.manaraa.com

The resulting $\boldsymbol{DFA_{t=5}}$ recognizes the language $\boldsymbol{L(DFA_{t=5})}$ and is shown in the transition diagram in figure 6-14. The three strings used as input have been used to create the following fully qualified deterministic finite automaton $\boldsymbol{DFA_{t=5}}$.

Each state has a transition to the trap state in the instance that an *invalid process invocation sequence* is encountered. The resulting machine is a fully qualified DFA with transitions specified as in figure 6-15.

	μ	p_1	p_3	p_2
$\underline{P_0}$	H	P_1	H	H
$\underline{P_1}$	H	P_1	P_3	P_2
$\underline{P_2}$	H	H	P_3	H
$\underline{P_3}$	H	H	H	H
H	H	H	H	H

Figure 6-15 $DFA_{t=5}$ **Transition Table**

After profiling $(p_1\, p_2\, p_3)$, $\boldsymbol{DFA_{t=5}}$ can profile the next *valid process invocation sequence*. If $\boldsymbol{DFA_{t=5}}$ encounters a previously unseen process invocation sequence such as $(p_1\, p_1\, p_2\, p_3)$, it can accept that sequence without the necessity of creating a new machine.

A new DFA is created after each new process invocation sequence is encountered in "profiling" mode. When "profiling" mode is turned off, the DFA has established steady state, no new processes are added to the PPT DFA. If we examine the $\boldsymbol{DFA_{t=5}}$ transition table created by the PPT profiling, we can see that other unseen *valid process invocation sequences* are accepted by the DFA, e.g. $p_1\, p_1\, p_1\, p_1\, p_3$ and $p_1\, p_1\, p_1\, p_1\, p_2\, p_3$. This does not present a problem, because the DFA has established that P_1 can invoke P_1, P_2 and P_3, and that P_1 can invoke P_1 any number of times. Whether or not P_1 should invoke P_1 more than once is a matter of program execution correctness, but is not a matter of authorizing P_1 to invoke P_1. The validity of the substring p_1p_1 has been established. An algorithm for building a PPT DFA is given in figure 6-16.

المنارة للاستشارات
www.manaraa.com

Algorithm for building a PPT DFA

Set ***whitelist*** = { } // set of known valid processes
Set $\boldsymbol{Q} = \{\boldsymbol{P_0}, \boldsymbol{H}\}$ // set of DFA states
Set $\boldsymbol{F} = \{\boldsymbol{P_0}\}$ // set of accepting states
Set $\boldsymbol{\Sigma} = \{\ \}$ //initialized alphabet
Set tempstring $\boldsymbol{w} = \boldsymbol{\varepsilon}$
Get input sting
For ($\boldsymbol{i}$=1; $\boldsymbol{i} \leq$ n; $\boldsymbol{i}$++)
    Set $\boldsymbol{w} = \boldsymbol{wp_i}$
    IF ((***w*** is not in ***whitelist***) and (expert validates ***w***))
Add $\boldsymbol{P_i}$ to $\boldsymbol{Q}$
        Add $\boldsymbol{P_i}$ to $\boldsymbol{F}$
        Add $\boldsymbol{p_i}$ to $\boldsymbol{\Sigma}$
        Add DFA transition $((\boldsymbol{P_{i-1}}, \boldsymbol{p_i})\boldsymbol{P_i})$
        Add ***w*** to ***whitelist***
ENDIF
ENDFOR

Figure 6-16: Algorithm for Building PPT DFA

6.4 Relation Between L_v and $L(DFA_t)$

Because the DFA is built incrementally, it is not known at any value time $\boldsymbol{t}$ whether $\boldsymbol{L(DFA_t) = L_v}$. However, the strings in the set called the *white list* are in the non-empty intersection $\boldsymbol{L(DFA_t)} \cap \boldsymbol{L_v}$. Because the *white list* is also built incrementally it cannot be claimed that the set of strings called the *white list* is exactly the set of stings $\boldsymbol{L(DFA_t)}$. Furthermore, it is not known whether the set $\boldsymbol{L(DFA_t)}$ merely form a non-empty subset of $\boldsymbol{L_v}$. $\boldsymbol{L(DFA_t)}$ could contain a set of strings that are not a subset of $\boldsymbol{L_v}$. That is, it is not known whether there exists another non-empty subset of strings both in $\boldsymbol{L(DFA_t)}$ and outside $\boldsymbol{L_v}$. More formally, it is not known whether $(\overline{\boldsymbol{L_v}} \cup \overline{\boldsymbol{L(DFA_t)}}) = \{\ \}$. This remains an open theoretical question and a topic for future research. The PPT model uses domain knowledge both to build $\boldsymbol{DFA_t}$ and to determine whether strings known to be in $\boldsymbol{L(DFA_t)}$ are also in $\boldsymbol{L_v}$. In this way the PPT model incrementally builds

www.manaraa.com

the non-empty intersection $\boldsymbol{L(DFA_t)} \cap \boldsymbol{L_v}$ called the *white list*. This open theoretical question is further discussed in appendix E.

المنارة للاستشارات
www.manaraa.com

Chapter 7: Implementation of the PPT Model

In this chapter the PPT DFA theoretical computational model is instantiated into a structure that can be implemented on a computer system in the form of a Program Pathing Trust Machine (PPTM). The transition table represented at the end of chapter 6 provides a map for moving the theoretical computation model to an implementation of the PPT model into a PPTM.

The PPTM will operate in two modes. One mode is the learning mode. Psuedo code for the learning mode was provided in figure 6-16. Psuedo code for the validation mode is provided in figure 7-1.

PPT DFA Validate Mode

Use DFA built in Learning mode
Set ***greylist*** = { } //holds invocations strings of unknown validity
Set current DFA state to P_0
Set tempstings $w = \varepsilon$
Get input sting of form $p_1p_2\ldots\ldots p_i\ldots p_n$ for $p_i \in \Sigma_\gamma$
FOR (***i***-1;***i*** $\leq$ ***n***; ***i***++)
    Set ***w*** = ***wpi*** //append pi to w
    IF (***w*** is not in ***whitelist***)
        IF(transiton ((***Pi-1***, ***pi***) is valid) and (***pi*** is in ***F***)) THEN
            Place ***w*** in ***greylist***
        ELSE
            Reject ***w*** and END
        ENDIF
    ENDIF
ENDFOR

Figure 7-1: Algorithm for PPT DFA in Validation Mode

The remaining sections of this chapter describe the issues associated with implementation of PPTM.

المنارة للاستشارات
www.manaraa.com

7.1 Alternatives for Implementing the Program Pathing Trust DFA

Implementing the PPTM using a transition table can be done in a number of ways. The application of the DFA dictates what implementation strategy should be used. Each application of the PPT DFA has its own requirements. For instance, in discretionary and mandatory access control the number of process invocation sequences mapped for access to a particular resource is small and therefore a small and simple mapping structure is sufficient. For an integrity trusted model, which is the focus of this research, all the valid process invocation sequences are mapped. Because a much larger number of processes and process invocation sequences are mapped, a different structure to implement the PPT DFA model is advisable.

7.1.1 PPT DFA Bit Map Implementation

For discretionary and mandatory access control applications, only a small number of process invocations need to be mapped. Take for instance the process invocation sequences where only Process P_3 is allowed to access resource X, using the invocation sequences $p_1\, p_1\, p_3$, $p_1\, p_2\, p_3$, and $p_1\, p_1\, p_2\, p_3$. The number of *normal processes* and *valid process invocation sequences* in the alphabet are very small in number and can be implemented in a very simple structure such as a bit map.

Process Invocations

Process	μ	p_1	p_2	p_3
P_0	0	1	0	0
H	0	0	0	0
P_1	0	1	1	1
P_2	0	0	0	1
P_3	0	0	0	0

Figure 7-2 Program Pathing Bit Map

Consider a small population of process invocations. For this population, a bit map can be used effectively for the mapping structure to represent the PPT DFA. Consider Figure 6-15, the

المنارة للاستشارات
www.manaraa.com

DFA transition table in chapter 6. The table is easily represented by the instantiation of the bit map structure in Figure 7-2.

Initially in the bit map structure all process invocations in the alphabet Σ are initialized to binary 0s - meaning all processes invoked end up in the trap state. Any transition that is marked as binary 0 is defined as a transition to the trap state H. As the DFA learns new processes and new valid invocations, a binary 1 is placed in the cells where the process in the row is authorized to invoke the process in the column. A binary 0 in a cell means move to the trap state, from which there is no escape. A binary 1 means that the process in the row can invoke the process in the intersecting column.

The Program Pathing bit map implementation uses the adjacency-matrix representation used in graphic structures [Sedg02]. Using an adjacency-matrix graph of n by n array of Boolean values, a Program Pathing bit mapping implementation of an DFA can be built using a small amount of storage, given a small number of processes. The advantage of using a bit map is that it allows for mapping every combination of invocations of the processes represented in the matrix. Looking up invocations and adding new invocations to the matrix is relatively simple, and computationally inexpensive.

Adding an invocation or invocation sequence entails adding new processes to the row and column and changing the binary 0s to 1s for the cells representing process invocations.

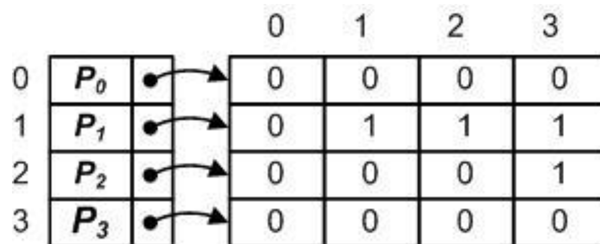

Figure 7-3: Adjacency-matrix

المنارة للاستشارات
www.manaraa.com

Figure 7-3 represents the adjacency-matrix for the bit map represented by figure 7-2 and figure 6-15.

Note in figure 7-3 that the first table is an array of process names with pointers to rows in the bit map. This is done so that the process name can be looked up. The relative offset into the process name table is the relative position of the process in the bit map, as indicated by the numbers along the rows and columns.

A disadvantage of using a bit map is that most operating systems are written in the C language, which has poor bit manipulation, and therefore coding a bit map in the C language is challenging. Another disadvantage of the adjacency-matrix bit map is that it may result in a wasting space. Mathematically, the adjacency–matrix is still more efficient with storage than the retrieval tree approach. For discretionary and mandatory access control applications the adjacency-matrix is sufficient, since these applications are interested in allowing only a few process invocation sequences for access to a particular resource.

Access control identifies the program pathing DFA in the access control list for the resource being restricted. Therefore in an access control system there are multiple program pathing DFAs identifying process invocation sequences for each resource needing program pathing controls. Although the adjacency-matrix bit map implementation would be sufficient for access control, it may not however be sufficient for intrusion detection or ensuring trusted systems where all process invocation sequences must be mapped. Therefore another implementation approach must be considered,

المنارة للاستشارات
www.manaraa.com

7.1.2 PPT DFA Adjacency-List Implementation

The mapping of all *valid process invocation sequences* is required for intrusion detection or to establish a trusted system. The adjacency-matrix bit mapping approach to program pathing would require too much memory to map all process invocations in a system. In such cases, the program pathing DFA mapping structure can be implemented in the form of an adjacency-list [Sedg02]. The adjacency-list approach implementation of the program pathing DFA removes the empty spaces in the adjacency-matrix by using linked lists. Although this approach uses more memory for smaller process invocation sequences, it ends up taking less space for larger process invocation sequences, particularly if the adjacency-matrix is sparsely populated. The benefit of the adjacency-list approach is that it can map a large diverse set of process invocation sequences more efficiently if there are a large number of different processes performing the invocations and little redundancy. Figure 7-4 illustrates the adjacency-list approach mapping the same $p_1\, p_1\, p_3$, $p_1\, p_2\, p_3$, and $p_1\, p_1\, p_2\, p_3$ invocation sequences used in Chapter 6, figure 6-15.

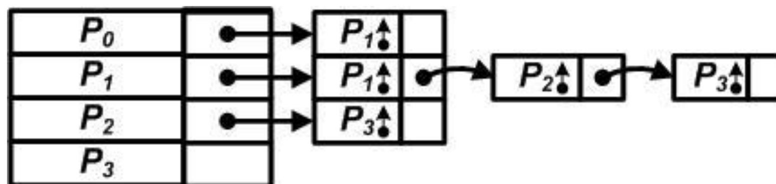

Figure 7-4: Adjacency-list

Although the adjacency-list mapping takes more memory per invocation, as the number of processes increases it takes less space when compared to the adjacency matrix. If p = the number of processes invocations in a sequence, then the adjacency-matrix uses p^2 space to implement the mappings whereas, the adjacency-index uses $p+L$ (where L = the count of linkages to invoking processes.)

Using an adjacency-list to implement the program pathing DFA is more efficient than earlier approaches because it does not have to map a process invocation multiple times if it appears in

www.manaraa.com

different invocation sequences, thereby saving space. The adjacency-list approach is more efficient than the adjacency-matrix in situations were the number of unique processes making process invocations is greater than 110 processes or 1.5K of storage. The adjacency-index method can represent 192 process invocation relationships in 1.5K, whereas the adjacency-matrix can only represent 110 processes but also represent all the possible process invocation relationships of those processes. The adjacency-matrix method is much more efficient in representing all the possible process invocation relationships between the processes in the matrix – there is no extra cost for representing a process invocation between processes already represented in the matrix. But adding a new process invocation relationship between a process already existing in the adjacency-list implementation always has a cost.

7.2 Measuring Implementation Structures

Preliminary results showed that the program pathing approach using an adjacency-matrix or adjacency-list can provide a more efficient and simpler mapping of process invocation sequences. Figure 7-5 shows a comparison between the adjacency-matrix, the adjacency-list and the retrieval tree approaches. The "Implementation Approaches" columns shows the number of processes each approach can represent give the "Memory" allocation.

Memory			Implementation Approaches		
Bits	Bytes	Storage	Adjacency Matrix	Retrieval Tree	Adjacency List
8,192	1.024	1K	90	85	128
8,388,608	1,048,576	1M	2,896	87,381	131,072
8,589,934,736	1,073,741,842	1G	92,681	89,478,486	134,217,730

Figure 7-5: Memory and Process Representation Comparisons

In 1K of storage, the adjacency-matrix approach can represent close to the same number of processes as the other two approaches, however its advantage is that it can represent more process invocations between the processes it represents. Both the retrieval tree and the

المنارة للاستشارات
www.manaraa.com

adjacency-list approaches can only represent the same number of process invocations as there are processes represented and adding new process invocations has a cost. For the retrieval tree approach, the cost is higher because each process invocation must identify every process that it invokes, even if it has been mapped before.

The adjacency-matrix approach is also efficient in creating and processing the mapping structure which allows for easy addition of processes and process invocations. Adding a process to the matrix can be done by adding a new row and column to the matrix. Validating if a process invocation is authorized is as easy as verifying that the process is represented in the matrix and that the cell in the matrix that represents the process invocation is set to a binary 1.

For larger mappings of multiple and more complex process invocation sequences the adjacency-matrix can become too large. For some applications, this disadvantage can be overcome by breaking up the program pathing DFA into multiple DFAs, like SELINUX does with its reference policies [Smal01b]. Each DFA represents a domain of *valid process invocation sequences*. The relationship between the DFA domains can be mapped in a higher level DFA of domain invocations. The SELINUX approach using domains makes the adjacency-matrix approach an optimal solution, where it can be applied. However, in the case where all the process invocation sequences must be mapped in a single DFA, the adjacency-list approach is a better implementation of the program pathing model. The approach allows for the most efficient use of storage.

The program pathing approach, regardless of the method used to implement it (adjacency-matrix or adjacency-list) provides a good alternative to previously tried methods, i.e. n-gram [Hof98] or retrieval trees [Amm98]. Even though both methods' DFA can recognize process invocation sequences not previously learned, this is not unlike a machine learning algorithm. If a

المنارة للاستشارات
www.manaraa.com

process invocation has been learned and identified as trusted, then it should be trusted in other process invocation sequences that lead to the same process invocation.

7.3 Coding Structures in PPTM

Instantiation of a PPT DFA entails creating data areas that support the adjacency-matrix. Developing an implementable PPT DFA using an adjacency-matrix is not just a matter of coding the structure, the structure has to be designed for maintainability. The design techniques used must be scalable and allow for ease of diagnostics. The Program Pathing Trust Machine (PPTM) was written in C language, the language of choice for most operating systems. Ideally, the PPTM would be integrated as a subsystem in the operating system's kernel. However, the PPTM is only a functional prototype to prove that an instantiation of a PPT DFA is possible and is capable of solving the problem. Further testing has to be performed in production ready systems to prove that the prototype is sufficient. The C language also provides the ability to use and maintain address pointers, which is useful in designing an implementation of the PPT DFA.

The PPTM prototype was not implemented into the system's kernel, but as a stand-alone application that creates the PPT DFA structure, for mapping *valid process invocations sequences* and for validating the authority of process invocation s to determine if they are *invalid process invocations*. Verification of the prototype was essential before attempting to make any modifications to the system's kernel. Complete exploration of issues concerning the modification of the system's kernel is a topic for future research (see chapter 9).

7.3.1 PPTM Basic Structure

This section describes the data areas created to realize a functioning PPTM. The data areas support an implementation of the PPT DFA described in chapter 6.

المنارة للاستشارات
www.manaraa.com

The main data area for the PPTM is the *anchor*, as shown in figure 7.6. The *anchor* data area is the communication vector that is an anchor point for all the PPTM's basic components. The anchor data area

Figure 7-6: PPTM Anchor Data Area

is made up of a length field, an eye catcher field and a number of address pointers. The fields are defined as follows:

len Length of the entire anchor data area. This field is initialized after the data area is allocated and is used to deallocate the data area when the application ends.

eyecat The eye catcher field is initialized with the ASCII text of "ANCHOR." The ASCII text allows a technician to quickly identify the anchor data area in a core dump of memory when diagnosing the application.

stkptr Address pointer to allocated memory data area called *stack*, which is a block of singly linked list cells each defined by the *scell* data area. *Scells* define the invoked processes.

sptr Address pointer of the next available unused *scell* in the *scell stack* data area.

snum Number of the next indexed *scell* available in the *scell stack* data area.

autptr Address pointer to the *automata* data area, where the names of all the encountered processes are recorded in an array of data areas called *acell*.

aptr Address pointer to the next unused *acell* available in the *acell automata* data area.

anum Number of the next indexed *acell* available in the *acell automata* data area.

The PPT DFA is made up of four sets of data areas: *automata, stack*, *acell* and *scell*.

المنارة للاستشارات
www.manaraa.com

These four data structures work together to instantiate the PPT DFA. The *automata* data area is the main structure of the PPT DFA; it is an array of the states in the DFA or the *normal processes*. The elements in the *automata* data area are *acell*s; each *acell* represents a state or *normal process* with reference to the DFA 5-tuple characteristics of the PPT DFA. $\boldsymbol{Q}$ = *automata* data area or $\boldsymbol{Q}$ = (*acell*[1], *acell*[2],…*acell*[*n*]). The alphabet $\boldsymbol{\Sigma}$ is represented by the elements in the *stack* data area called *scell*s. They represent transitions or *valid process invocations*. $\boldsymbol{\Sigma}$ = *stack* data area or $\boldsymbol{\Sigma}$ = (*scell*[1], *scell*[2],…*scell*[*n*]).

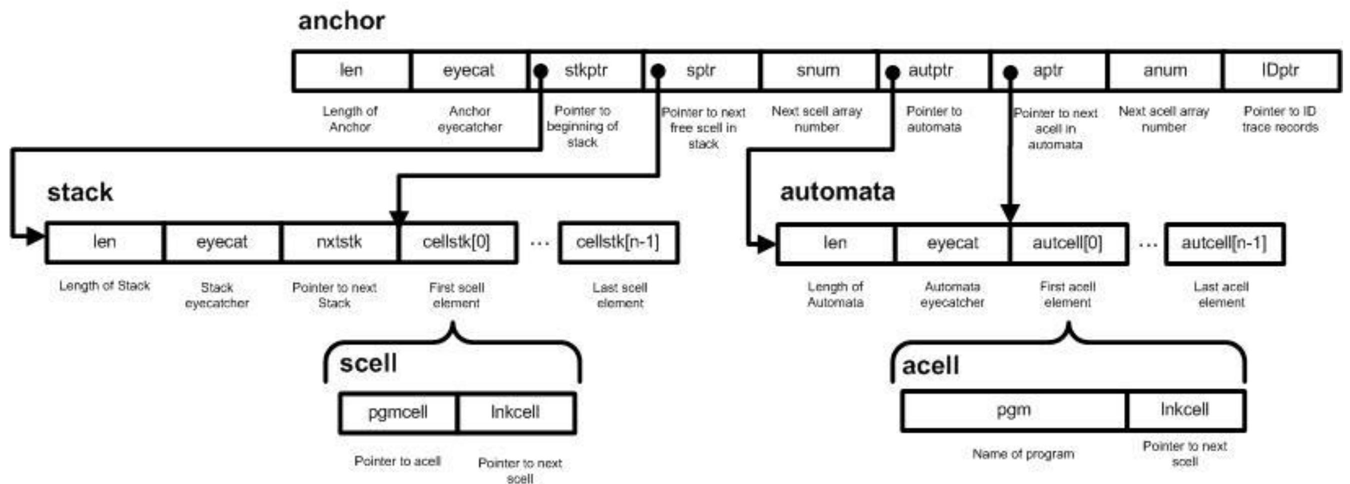

Figure 7-7: Automata Data Structures

7.3.2 How the PPTM works

The prototype is initialized by allocating the *anchor*, *stack* and *automata* data areas. These blocks of memory all have a length (len) and eye catcher (eyecat) initialized so that the end of each data area can be determined and so that the data area can be easily found in a memory core dump. The size of the data area blocks in the prototype is arbitrarily determined by an internally defined variable, however in a production-quality implementation the data areas can be allocated dynamically depending upon the size of the stored DFA recorded (for instance the number of *acell*s allocated in the *automata* data area are exactly the number of processes recorded). The

www.manaraa.com

size of the data area is important for the purposes of de-allocating the memory at a later time, and for reading a memory core dump if necessary.

Once the main data areas are allocated and initialized the PPTM reads the *valid process invocation sequences* needed to populate the automata data area. Populating processes into the cells is a matter of recording processes as they are encountered when the PPTM is in recording mode – in the case of the prototype, this means reading the "train" file. In recording mode, all process invocation sequences are assumed to be *valid process invocation sequences* and are recorded in the automata.

After all the recorded processes are loaded into the *automata* data area, the PPTM prints out the automata structure into an "autotrace" file. This is to allow auditing of the automata data area created by the PPTM application. Using the example of the *valid process invocation sequences*

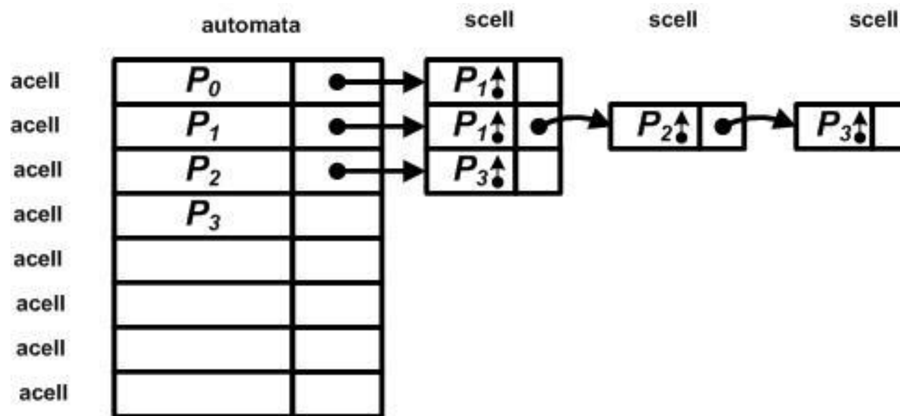

Figure 7-8: "automata" Data Area Containing the Process Invocation Sequences

$p_1p_1p_3$, $p_1p_2p_3$, and $p_1p_1p_2p_3$, (described in chapter 6), the automata data area would look as illustrated in figure 7-8.

The PPTM *automata* data area is interpreted from the in core memory version and translated into a text file. The resulting "*automtrace*" file is a representation of the automata data area in grammatical form, indicating that the process on the left can call either the processes P_1, P_2 *or* P_3.

المنارة للاستشارات
www.manaraa.com

```
Automata Trace
Caller Process -> Called Process | Called Process
------------------------------------------------------
S -> ProcessP1
ProcessP1 -> ProcessP1 | ProcessP2 | ProcessP3
ProcessP2 -> ProcessP3
ProcessP3
```

Figure 7-9: Format of the "*automtrace*" file

The PPTM then validates any process invocation sequences it encounters against the *valid process invocation sequences* it has recorded in the *automata* data area. The PPTM is now in validate mode. Any process or process invocation that is encountered but not authorized is recorded as an error. For the sake of the prototype, all unauthorized processes or process invocations that are not valid are recorded in the "*auditfile*" file. In validate mode, if the PPTM encounters a process that has not been recorded in the *automata* data area, the PPTM writes an error message to the "*auditfile*" file:

`Called process [ID2] - [process name] invalid.`

If on the other hand the PPTM encounters a process invoking a process that is a *normal process*, but is not authorized to invoke, then the PPTM writes another error message to the "*auditfile*" file:

`It is invalid for PID [ID3] - process [process name 1]`[6] `to call process [process name 2]`[7].

As a process is validated, the PPTM writes out the process invocation process so that it can be audited later. If the audited process invocation sequence was validated incorrectly, a technician can correct the PPTM by submitting the audited process invocation sequence to be recorded in the *automata* data area.

[6] `[Process name 1]` refers to the process attempting to invoke process 2
[7] `[Process name 2]` refers to the process being invoked by process 1

المنارة للاستشارات
www.manaraa.com

Chapter 8: Development and Test Results

Development and testing of the PPTM prototype was done on a Dell Latitude C640 with an Intel Pentium 4, 1.80 GHz, 500 MB RAM and 30GB hard drive hardware, running Linux Red Hat Fedora Core 6. It was also tested on a Dell Latitude D600 with an Intel Pentium M, with 1.6 GHz, 2 GB RAM and 60 GB hard drive hardware. This system ran Windows XP with service pack 3. The code was additionally tested on a MacBook 5.2 with an Intel Core 2 Duo, 1.23 GHz, 4GB RAM and 160GB hard drive, running OS X 10.6.5. The code was recompiled on each of these systems and ran without any problems.

8.1 Development

Implementing the PPTM involved making some decisions concerning the operating system and the programming language to be used. Given that the PPTM would eventually reside in the OS kernel, it was decided to use C language and given that the PPTM has to be used in multiple OS environments, this research chose GNU's **gcc** compiler, since it supports multiple OS environments. The source code was transferred to two other computers running different OS `systems for testing.

The source code was placed in a development directory. The system path was positioned to the source code in the development directory using the `PATH=$PATH:.` command. The `gcc PPTM.c -o PPTM` command was used to create an executable program called PPTM. A copy of the PPTM source code can be found in Appendix A.

المنارة للاستشارات
www.manaraa.com

8.2 Unit Testing and Debugging

Testing the PPTM was a challenge because it proved difficult to acquire test data from production or quality assurance (QA) critical servers that ran processes that needed to be protected. In order to secure sufficient test data from critical servers, this research would have to demonstrate that the algorithms perform and function as designed. Initially it was important to verify that the code performed the functions it was designed to perform properly. A series of unit tests were conducted to verify each function.

8.2.1 Test Reading Training Data and Building the Automata Structure

Mock training data was used to test the PPTM program's ability to read in the data and store it in the automata. The training data was sparse to make it easy to debug and was crafted to test the different features of the automata structure. The implemented automaton is made up of acells and scells. An acell is a data area that represents a process, or in computational theory vernacular, a state. The scell data area represents a transition to a state. The test training data verified that both acells and scells were created properly. If an acell for a process already existed, another acell would not need to be created and the acell for the process would be re-used.

As a process invokes other processes, scells representing the invoked processes are linked to the invoking process's acell. The scell also links to the invoked process's acell, if it exists, otherwise an acell is created for the invoked process and the linked scell in the invoking process acell linklist points to the invoked process's scell. If a process's scell is already linked to an invoking processes acell, then a duplicate scell does not need to be linked to the acell. The scell would be re-used.

المنارة للاستشارات
www.manaraa.com

After all the crafted mock training data was read and modeled in the PPTM automaton, the PPTM code printed out the automaton representation to an "*automtrace*" file. The "*automtrace*" file was used to verify that the training data was represented properly in the automaton and could be used for debugging later.

8.2.2 Verify Data Against the Profiled Training Data in the Automata

Once the automata machine was loaded with the mock test data, and the "*automtrace*" file verified that the automata structure was successfully built, the PPTM prototype was then tested to see if it could be used to verify process invocations. The validation testing determined if the automata could identify the process invocation sequences that the PPTM loaded into the automata and determined if it was able to identify *invalid process invocation sequences*. The testing was intended to verify that the automaton could identify the following *invalid process invocations*:

1. An *abnormal process* tries to invoke a process.
2. A *normal process* tries to invoke an *abnormal process*.
3. A *normal process* tries to invoke a *normal process* that it is not authorized to invoke.

8.3 System Testing

In absence of available test data, mock test data was created programmatically. The *Tstdata* program was created to generate test data to test the PPTM program. See Appendix A for the *Tstdata* souce code. The *Tstdata* program randomly generates system test data to test the PPTM program and to find any problems that the simulated hand crafted test data in the functional testing did not reveal.

المنارة للاستشارات
www.manaraa.com

8.3.1 Tstdata – Random Test Data Generator

The *Tstdata* program randomly generates process names and process invocation sequences. The Tstdata has three defined constants in the *Test.h* file which control the volume and characteristics of the test data.

1. *NoProcess* – Defines the number of processes the *Tstdata* can choose from.
2. *NoIDs* – Defines the number of process invocation sequences to generate.
3. *NoProString* – Defines the maximum number of processes that can be in a process invocation string.

The *Tstdata* program generates *NoIDs* number of process invocation sequences, choosing 1 to *NoProString* number of processes in a sequence. The number of processes in a sequence is randomly generated for each sequence, so each may have a different number of processes. Each process invocation sequence is assigned a UID number 1 through *NoIDs*, e.g., {UID1, UID2, …,UID*n*}. Each process added to a process invocation sequence is randomly chosen from a number 1 to *NoProcess*. The processes are assigned a name based upon the random number generated with an M preceding it, e.g., {M1, M2, M3, …, M*n*}.

The test data generated is in the following format:

```
UID1  S  M291
UID1  M291  M876
UID1  M876  M97
UID2  S  M79
UID2  M79 …
```

Note that every process invocation sequence starts with the start state ***S***, which is equivalent to the P_0 start state described in the theoretical model in chapter 6.

المنارة للاستشارات
www.manaraa.com

Each line in the test data represents a process invocation, identifying the process sequence it belongs to (indicated by the UID*n*), the invoking process, immediately after the process sequence number and the invoked process, right after the invoking process name.

The variables in the *Test.h* file can be changed to simulate various process invocation sequence scenarios. The *Tstdata.c* program must then be re-complied using the GNU **gcc** compiler to accept the changes made in the *Test.h* file. The *Tstdata* program is then executed to generate test data that is written to the *Train* file, so it can be used as training input to the PPTM program.

8.3.2 Performance Testing the PPTM prototype

System testing focused upon establishing the performance baselines for building the automata and validating process invocation sequences against training data loaded into the automata. Unlike the functional tests, this data was larger in volume and more complex. The test data was generated automatically using the *Tstdata* program. The table in figure 8-1 shows the system testing and the parameters used to test the PPTM prototype.

Figure 8-1 is the results from the system test. The table scells and acells represent the memory allocation of the PPTM data structures to so that the amount of memory necessary to represent the process invocation sequences using either the adjacency-matrix or the adjacency – list can be determined. The test provides statistics to verify the amount of memory that PPTM needs for each approach. It is important to know the memory requirements of PPTM when it runs on critical servers so as to prevent system resources from being over utilized by PPTM. Since PPTM is to be implemented into the kernel to intercept processes being scheduled for dispatching to the CPU, the PPTM structures should not use up too much RAM.

المنارة للاستشارات
www.manaraa.com

The result of system testing uncovered some problems not discovered in the functional testing, such as buffer overflows (which happened when a large volume of test data was presented to the PPTM). Further, the system test identified the fact that an automaton becomes denser as the invocation sequences invoke more processes.

System testing demonstrated that the PPTM could handle a large number of processes and sequences. The tests performed as expected, the PPTM profiled all the test data and identified the process invocations it should have indentified as invalid. No anomalies were found during testing of the validation phase of the PPTM

# of sequences	# of processes	Max # of processes per sequence	acells	scells	Max number of possible invocation combinations	Memory allocation for matrix in MB	Memory allocation for link-list in MB	percentage density
10	10	10	10	40	121	0.000320	0.0009	33.06%
10	100	50	93	253	10,201	0.004054	0.0067	2.48%
10	1,000	100	434	567	1,002,001	0.132692	0.0219	0.06%
100	10	10	10	116	121	0.000320	0.0021	95.87%
100	100	100	100	3,979	10,201	0.004268	0.0638	39.01%
100	1,000	100	997	5,460	1,002,001	0.149874	0.1137	0.54%
1,000	10	10	10	118	121	0.000320	0.0021	97.52%
1000	100	100	100	10,074	10,201	0.004268	0.1568	98.76%
1000	1,000	100	1,000	48,438	1,002,001	0.149965	0.7696	4.83%
1000	10,000	100	9,943	49,569	100,020,001	12.226750	1.0598	0.05%
10,000	10	10	10	116	121	0.000320	0.0021	95.87%
10,000	100	100	100	10,170	10,201	0.004268	0.1582	99.70%
10,000	1,000	100	1,000	392,882	1,002,001	0.149965	6.0254	39.21%
10,000	10,000	100	10,000	494,121	100,020,001	12.228489	7.8449	0.49%
100,000	10	10	10	117	121	0.000320	0.0021	96.69%
100,000	100	100	100	10,164	10,201	0.004268	0.1581	99.64%
100,000	1,000	100	1,000	994,279	1,002,001	0.149965	15.2020	99.23%
100,000	10,000	100	10,000	2,430,739	100,020,001	12.228489	37.3953	2.43%
100,000	100,000	100	100,000	5,025,487	10,000,200,001	1,195.168495	79.7346	0.05%

Figure 8-1 System Test Results

The high mark testing was 100,000 processes in 100,000 process invocation sequences averaging 50 processes in each sequence. All 100,000 processes were used in the invocation test, and over 5 million process invocations were profiled. Memory utilization for the PPTM using the adjacency-list and the adjacency matrix were calculated. The amount of allocated

المنارة للاستشارات
www.manaraa.com

RAM needed to represent the PPTM adjacency-list structure was about 80MB versus over 1 GB using the adjacency-matrix. The density of the invocations was measured to determine the point at which the adjacency-matrix began out performing the adjacency-list structure.

For tests where the number of processes were between 10 and 1000, there was no significant difference between the two structures. As the number of processes increased, the adjacency-list began allocating less memory than the adjacency-matrix. This trend continues until the number of process invocations grows and begins filling the adjacency-list. When the adjacency-list structure starts to converge to around 40% of the maximum number of possible invocation combinations, the adjacency-matrix starts to become the more optimal structure for conserving memory. It is, however, unlikely that process invocations would reach a 40% density. It would mean that if there were 10,000 processes, there would be 100 million process invocation possibilities and that the system would have to make 40 million of process invocations for the adjacency-list to be suboptimal. The tests suggest that the adjacency-list is the preferred structure to use of system integrity process invocation authorization.

المنارة للاستشارات
www.manaraa.com

Chapter 9: Future Research

9.1 Implement PPTM into the Operating System's Kernel

Implementation of PPTM requires that it be installed in the OS kernel so that it can validate program path sequence of process invocations. Of course, if the PPT model were to be adopted as part of an existing access control system, the OS intercept problem would be partially solved, as most access control systems already have intercepts in the OS.

Once the PPTM system is implanted into the kernel, it can monitor every process that is loaded for execution in real time. The PPTM system is able to verify all process invocation sequences. A fully functional PPTM subsystem must be developed with all the user interfaces and options to enable the PPTM prototype to function in a production environment.

9.2 Testing

The PPTM has been tested with mock data. However additional testing with real data is necessary once the PPTM has been implemented in a system as described above. All the features in the PPTM implemented in the kernel described in section 9.1 must be tested and with a number of application scenarios. A number of known applications have to be tested in combination and separately to determine if the system can identify and distinguish between valid and invalid process sequences. To understand how exactly the PPTM system works, the system must be tested for performance as well as accuracy. Comparing the results of PPTM to other process invocation sequence models is difficult, due to lack of a standard test bed, but comparative evaluations should be made so far as possible. A useful extension of this research may be the development and proposal of a standard test bed made available to other researchers to facilitate comparisons of approaches.

المنارة للاستشارات
www.manaraa.com

9.3 Process Authentication

Process authentication is important for determining that a process is the process it purports to be. Future research will determine whether the PPT model can en extended to include authentication. Further discussion on this issue is presented in appendix D.

9.4 The Validity of Inferred Process Invocation Sequences

One may make the assumption that by authorizing individual process invocations the PPT model can infer *valid process invocation sequences* it has not yet encountered. This assumption may allow the PPT DFA to accept sequences that are invalid. Further research should aim at determining whether some members of $L(DFA_t)$ are not in L_v. Further discussion on this issue is presented in appendix E.

المنارة للاستشارات
www.manaraa.com

Chapter 10: Conclusion

This research has identified a model to validate process invocations in order to prevent the execution of unnecessary processes that steal CPU cycles or otherwise interfere with production processing. Unlike intrusion prevention, the significance of program pathing is to keep *normal processes* from being invoked at inappropriate times, as well as to keep malware from running. The goals of the program pathing model are to be scalable to a production environment, and to take relatively little time and knowledge to implement and maintain.

The first principle of engineering is to analyze and understand the problem to be solved. Rigorous analysis of a problem often yields a good solution, and one that is not overly complex. A simpler solution is easier to manage, thus better positioned to perform optimally. This research has analyzed the problem of validating *process invocation sequences* using a computational theory approach.

10.1 Computational Theory Approach to Validating *Process Invocation Sequences*

Bell and LaPadula stated that it was important to "bridge the gap between general theory and practical problem solving" [Bell73]. And it is important to engage theoretical modeling in the problem solving process. This research has used computational theory to define and analyze the problem. Representing the *process invocation sequence* problem symbolically and examining it in the context of computational theory has enabled a more precise definition of the problem. Computational theory has focused the problem, allowing the solution to emerge from problem analysis.

www.manaraa.com

10.1.1 Required Computational Power

It was determined that a finite state automaton (FSA) has sufficient computational power to solve the problem of mapping process invocation sequences. In the DFA model chosen *valid process invocations* are mapped to verify the authority of each process to invoke or be invoked by another process. This technique assumes that all process invocations are invalid unless registered in the DFA.

To take into account *abnormal processes* from the alphabet Σ_β the automata had to define a new variable symbol μ. The symbol μ deviates from traditional automata theory. Traditional automata theory does not use variables in the alphabet. There is literature to suggest that a variable of indeterminate or unknown inputs in a transition might be acceptable [Buch60a] [Buch60b] [Elie74]. However, the present research did not further pursue these more computationally complex approaches because the DFA used here has sufficient computational power.

10.1.2 Translating Theory into Solutions

When dealing with even simple computational theory it is difficult to translate theory into implementation. There are some tools that allow researchers to convert regular expressions into implementation, such as lex and yacc [Levi95], but these are scripting languages and not applicable to the PPTM. The PPTM must be implemented into the kernel. This research has identified two possible data structures to implement an DFA – adjacency-matrix and adjacency-list [Sedg02]. Adjacency algorithms are graph algorithms, and the DFA is represented as a graph.

The adjacency-matrix (or bit map) fulfills all the requirements necessary to implement an DFA state transition table. The adjacency-matrix algorithm performs well if there are a large

المنارة للاستشارات
www.manaraa.com

number of input transitions. For state transition tables that have fewer input transactions, an adjacency-list (or link-list) is more memory efficient. The optimal implementation of the DFA depends upon the constraints dictated by the problem and the calculated memory requirements required to represent the DFA in implementing each of the adjacency algorithms.

10.2 Impact upon the Program Pathing Problem

This research's approach to the program pathing problem (process invocation sequence problem) has been to look for a simple solution. Instead of concentrating on mapping entire process strings or patterns, focus has been upon a process's authority to invoke another process. This simpler solution enables a DFA to profile invocation sequences more easily, as the invocation sequences are built from individual mappings of authorized process invocations. Other solutions have tried to map either the whole invocation sequence or substrings of that sequence.

Although the program pathing model prototyped has been functionally tested and system tested, it still needs to be embedded, implemented, and tested in an actual operating system. As stated in chapter 9, the real test for the model is for PPTM to be implemented without a lot of effort and to run effectively on a production system. This has to wait for future research and an institutional partner willing to spend the time testing.

10.2.1 Mapping Process Authority to Invoke Processes

The PPTM maps running process invocations as they are encountered to build the invocation sequences to be authorized. Instantiating the DFA into a series of linked lists of process invocations simplifies the mapping of sequences. A process cannot invoke another process unless the process that invoked it in turn was authorized to be invoked by another invoking

المنارة للاستشارات
www.manaraa.com

process. The valid invocation sequences are implied to be authorized, because only *valid process invocations* are allowed to be executed.

An advantage of the program pathing model is that it can populate the DFA linked list as process invocations are encountered and profiled from a running system. There is no need to edit or write a policy language to map process invocation sequences. If a process invocation sequence is encountered that has not been profiled by the program pathing DFA, the audit record identifying the encountered *process invocation* can be used to update the DFA, and capture that invocation as valid.

False negatives (i.e., prevention a process from being invoked when it should) can be prevented by placing the PPTM into Warn mode. Warn mode allows processes that may be critical to the operation to continue to be invoked, but alerts administrators that there is a potential false negative and something must be checked.

10.2.2 Mode Characteristics of Some Process Invocations

One characteristic of program pathing is that some process invocation sequences may be valid at one time but not another. For instance, it is not valid to run maintenance processes during production times. There may be processes that should run at scheduled times, and only at those times. The program pathing model must be able to handle cases where a sequence is valid at one point in time and invalid at another. The program pathing DFA can deal with these cases by (1) turning off the PPTM to allow the usually invalid processes to be invoked, or (2) switching the program pathing DFA to another DFA to allow the processes to run PPTM, or (3) including a time window as part of each process invocation symbol in Σ.

المنارة للاستشارات
www.manaraa.com

10.3 Potential Use of the Program Pathing Trust Model

The Program Pathing Trust Model is not intended to be a stand-alone function. It is intended to be a component of a larger security system as are the original program pathing in ACF2® [ACF99] and the program controls in SELinux [Mcca05]. That is to say, it is not intended to be used as a stand-alone function such as an intrusion prevention system (like Symantec's Critical System Protection [Suma05]).

10.3.1 Program Pathing in an Access Control System

The original use of program pathing was in a discretionary access control system (ACF2® [ACF99].) Its purpose was to verify that a user's access to a resource was granted but only through particular process invocation sequences. Any access to the resource outside the *valid process invocation sequence* was considered inappropriate and was denied. User authentication and user authorization to the resources is handled by the access control system, and the process invocation sequence validation could be handled by the Program Pathing Trust Model working as a component of the access control system.

Using the Program Pathing Trust Model to validate process invocation sequences would prevent a user from accessing data through any means but an authorized process, thereby adding a more secure dimension to access control. Restriction of user access to data in this way could help prevent unauthorized copying or leaking of data.

10.3.2 Program Pathing in a System Integrity System

A form of program pathing has been used in SELinux [Mcca05], and falls into some of the same pitfalls as the ACF2®'s implementation. It identifies the process invocation sequences allowed within a domain, and writes these relationships into a security policy using the SELinux

المنارة للاستشارات
www.manaraa.com

policy language. SELinux also validates access to other domains, resources, security labels and levels, etc. SELinux is a mandatory access control and system integrity system that could use the Program Pathing Trust Model in the same manner as discretionary access control. In the case of mandatory access control, a user with a security label could access a resource with the same security label but would only be allowed access it using specific process invocation sequences. SELinux has redefined mandatory access control to include system integrity [Mcca05].

The Program Pathing Trust Model can be used in the SELinux subsystem to define the processes within a domain. The Program Pathing Trust Model would remove the need to define each process manually in a security policy – thereby simplifying SELinux's implementation.

المنارة للاستشارات
www.manaraa.com

References

[ACF99] CA-ACF2 Systems Programmer Guide CA-ACF2 Release 6.3, September 1999

[ACF89] ACF99@RB module from CA-ACF2® Release 6.2, Computer Associates, International, 1989

[AFM99] Assurance in the Fluke Microkernel: Final Report, Secure Computing Corporation, Contract no. MDA904-97-C-3047, April 1999.

[Aho75] Alfred Aho, Margaret Corasick, "Efficient string matching: An aid to bibliographic search", Communications of ACM, Volume 18, Issue 6, pages 333-340, 1975.

[Aho97] Alfred Aho, "Algorithms for finding patterns in strings". In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, volume A, chapter 5, pages 255-300. MIT Press, 1994.

[Aho00] Alfred Aho, Jeffery Ullman, Foundations of Computer Science C Edition, Computer Science Press, NY, 2000.

[Amm98] Glenn Ammons, James Larus, "Improved data-flow analysis with path profiles," ACM SIGPLAN Notices, Volume 33, Issues 5 (May 1998), pp. 72-84.

[Appf04] The AppFire Suite for Host Intrusion Prevention: Technical White Paper, PlatformLogic, www.platformlogic.com (website no longer available see citation below)

[Ball92] Thomas Ball, James Larus, "Optimally Profiling and Tracing Programs," Proceedings of the 19th annual ACM symposium on Prinicples of Programming Languages, p.59-70, Albuquerque, NM, Jan. 19-22, 1992..

[Ball96] Thomas Ball, James Larus, "Efficient path profiling," Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, p.46-57, December 2-4, 1996, Paris, France.

[Bar61] Y. Bar-Hillel, M. Perles, and E. Shamir, "On formal properties of simple phrase-structure grammars". *Zeitschrift für Phonetik, Sprachwissenschaft, und Kommunikationsforschung* 14: 143–177, 1961.

[Bark80] Stephen Barker, The Elements of Logic, Third Editon, McGrall-Hill, NY, 1980.

[Beak69] George C. Beakley, Donovan L. Evans, Deloss H. Bowers, Careers in Engineering and Technology, Macmillan Company, 866 Third Avenue, New York, New York 10022, Page 448, 1969.

المنارة للاستشارات
www.manaraa.com

[Bell73a] Bell, D E, LaPadula, L J. "Secure Computer Systems' Mathematical Foundations," ESD-TR-73-278, vol. 1, ESD/AFSC, Hanscom AFB, Bedford, Mass., Nov. 1973 (MTR-2547, vol. 1, MITRE Corp., Bedford, Mass.)

[Bell73b] Bell, D. E., LaPadula, L. J "A Secure Computer Systems' A Mathematical Model," ESD-TR-73-278, vol. 2, ESD/AFSC, Hanscom AFB, Bedford, Mass., Nov. 1973 (MTR-2547, vol 2, MITRE Corp., Bedford, Mass.}

[Bell74a] Bell, D. E. "Secure computer systems: A Refinement of the Mathematical Model," ESD-TR-73-278, vol. 3, ESD/ AFSC, Hanscom AFB, Bedford, Mass, April 1974 (MTR 2547, vol. 3, MITRE Corp., Bedford, Mass.}.

[Bell74b] Bell, D. E, LaPadula, L. J. "Secure Computer Systems. Mathematical Foundations and Model," M74-244, MITRE Corp, Bedford, Mass., Oct, 1974.

[Bell76] D. E. Bell, L. J. LaPadula, "Secure Computer Systems: Unified Exposition and Multics Interpretation," ESD-TR-75-306, Project 522B, MITRE Corporation, Deputy for Command and Management Systems, USAF, Contract No. F19628-76-C-0001, March 1976.

[Biba77] K. J. Biba, "Integrity Considerations for Secure Computer Systems," ESD-TR-76-372, Project 522B, MITRE Corporation, Deputy for Command and Management Systems, USAF, Contract No. F19628-76-C-0001, April 1977.

[Bish03] Matt Bishop, "What Is Computer Security?," *IEEE Security and Privacy*, vol. 1, no. 1, pp. 67-69, Jan. 2003, doi:10.1109/MSECP.2003.1176998.

[Bove02] Daniel Bovet and Marco Cesati, Understanding the Linux Kernel, Second Edition, O'Reilly & Associates, 20

[Bran88] Branstad, M.; Tajalli, H.; Mayer, F. Aerospace Computer Security Applications Conference, 1988., Fourth Volume , Issue , 12-16 Dec 1988 Page(s):362 – 367

[Bre89] David Brewer and Michael Nash, "The Chinese Wall Security Policy," p. 206, 1989 IEEE Symposium on Security and Privacy, 1989

[Brzo62] J.A. Brzozowski. "Canonical Regular Expressions and Minimal State Graphs for Definite Events," Mathematical Theory of Automata, Vol. 12, MRI Symposia Series, pp. 529-561, Polytechnic Press, Polytechnic Institute of Brooklyn, NY, 1962.

[CC04] Common Criteria for Information Technology Security Evaluation Part 2: Security functional requirements, January 2004, Version 2.2 CCIMB-2004-01-002

[CCA08] CA Access Control, Protecting Server Resources with CA Access Control, Technical Brief: CA Access Control, 2008

المنارة للاستشارات
www.manaraa.com

[Chri08] Christian Christiansen, IDC White Paper - Server Resource Protection: A Critical Element of IT Security, Interactive Data Corporation, July 2008

[Clar87] D. D. Clark, and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," IEEE Security and Privacy Symposium, p. 184-194, April 1987.

[Copi78] Irving Copi, Introduction to Logic, Fifth Edition, MacMillian Publishing, NY, 1978.

[CSI03] Computer Security Institute, "2003 CSI/FBI Computer Crime and Security Survey," (2003).

[CST72] Computer Security Technology Planning Study, Deputy for Command and Management Systems, HQ Electronic Systems Division, ESD-TR-73-51, Vol. II, October 1972.

[CIP05] Critical Infrastructure Protection: Department of Homeland Security Faces Challenges in Fulfilling Cybersecurity Responsibilities, United States Government Accountability Office, Report to Congressional Requesters,GAO-05-434, http://www.gao.gov/cgi-bin/getrpt?GAO-05-434, May 2005.

[Dahl94] Robert Dahlberg, Personal Experience as a Software Developer on CA-ACF2 working on various aspects of the product including program pathing, SKK/UCCEL/Computer Associates, Chicago, IL, 1984 – 1994

[Denn87] Denning, D.E. "An Intrusion-Detection Model, Software Engineering," IEEE Transactions on Volume SE-13, Issue 2, Feb. 1987 Page(s): 222 - 232

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *Nuerische Mathematik*, 1:269-271, 1959.

[DoD85] Department of Defense Trusted Computer System Evaluation Criteria, DoD 5200.28-STD, Library no. S225,711 December 1985

[Eete08] Michael van Eeten, Johannes Bauer, Economics of Malware: Security Decisions, Incentives and Externalities, Organization of Economic Co-operation and Development, Directorate for Science, Technology, and Industry, Paris, France, 2008.

[Elm90] J. L. Elman, "Finding Structure in Time", Cognitive Science, 14:179-211, 1990.

[Fedo06] Fedora (2006). Retrieved 05/15/2006: http://www.fedora.info/

[Feng03] Henry Hanping Feng , Oleg M. Kolesnikov , Prahlad Fogla , Wenke Lee , Weibo Gong, "Anomaly Detection Using Call Stack Information," Proceedings of the 2003 IEEE Symposium on Security and Privacy, p.62, May 11-14, 2003

[Fire03] Donald Firesmith, Engineering Security Requirements, Journal of Object Technology, vol. 2, no. 1, January-February 2003, pages 53-68.

المنارة للاستشارات
www.manaraa.com

[FISM02] ''Federal Information Security Management Act of 2002", H. R. 2458—48,

[FISM08] "Federal Information Security Management Act of 2008", S.3474 .

[Ford97] Bryan Ford, Kevin Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, Jeff Turner, "*The Flux OS Toolkit: Reusable Components for OS Implementation*," The Sixth Workshop on Hot Topics in Operating Systems, Cape Cod, MA, USA, page(s): 14-19, May 5-6 1997

[Forr96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, Thomas A. Longstaff, "*A Sense of Self for Unix Processes*," sp, pp.0120, 1996 IEEE Symposium on Security and Privacy, 1996.

[Gass88] Morrie Gasser, Building a Secure Computer System, New Your: Nostrand Reinhold, 1988.

[Gau03] Hugh G. Gauch, Scientific Method in Practice, Cambridge University Press, 2003

[Gho00] Anup K. Ghosh , Christopher Michael , Michael Schatz, "*A Real-Time Intrusion Detection System Based on Learning Program Behavior*," Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection, p.93-109, October 02-04, 2000

[Gogu82] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," IEEE Symposium on Security and Privacy, 1982

[Grah72] G.S. Graham and P. J. Denning,"Protection – Principles and Practice," AFIPS Conference Proceedings, Vol. 40, Spring Joint Computer Conference, Montvale, New Jersey, 1972.

[Harr03] Shon Harris,"CISSP Certification Exam Guide", 2nd Edition, McGraw-Hill, 2003.

[Hart05] Hart, James L. M., Captain, USAF, "An Historical Analysis of Factors Contributing to the Emergence of the Intrusion Detection Discipline and its Role in Information Assurance", Thesis, AFIT/GIR/ENV/05M-06, Department of the Air Force Air University, Air Force Institute of Technology, 2005.

[Hick07] Boniface Patrick Hicks, "Secure System Development Using Security-Typed Languages," PhD Dissertation, Pennsylvania State University, Department of Computer Science and Engineering, December 2007.

[Hof98] Steven A. Hofmeyr, Stephanie Forrest, Anil Somayaji, "Intrusion Detection Using Sequences of System Calls," Journal of Computer Security, Volume 6, Number 3 / pp.151 – 180, 1998.

[Hogl05] Greg Hoglund, James Butler "Rootkits: Subverting the Windows Kernel," Addison-Wesley, 2005.

المنارة للاستشارات
www.manaraa.com

[Hopc01] John Hopcroft, Raheev Motwani, Jeffery Ullman, "Introduction to Automata Theory, Languages, and Computation," 2nd Edition, Addison-Wesley, 2001.

[IBM08] SMP/E V3R1.0 for z/OS and OS/390 Reference, SA22-7772-01, Online Library, available on the IBM Online Library Omnibus Edition: MVS Collection CD-ROM, SK2T-0710, 2008.

[IBM09] z/OS TSO/E Bookshelf, GA22-7489-12 Online Library available on the IBM Online Library Omnibus Edition: z/OS DVD Collection (SK3T-4271), 2009

[Ietf] Internet Engineering Taskforce (IETF), RFCs 2903 2904, 2905, 2906, www.ietf.org

[ISO96] Security Frameworks for Open Systems: Access Control Framework, Technical Report ISO/IEC 10181-3, ISO, 1996

[Jaeg04] Trent Jaeger, Anthony Edwards, Xiaolan Zhang, "Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework," ACM Transactions on Information and System Security (TISSEC),Volume 7 , Issue 2 (May 2004), Pages: 175 - 205

[Jaeg05] Trent Jaeger, "Clark-Wilson Integrity as a Security Goal for SELinux Policies," IBM TJ Watson Research Center, USA, SELinux Symposium, 2005
http://www.selinux-symposium.org/2005/presentations/session5/5-2-jaeger.pdf

[Ju07] Hu Jun, Shen Changxiang, "An Information Flow Security Model to Trusted Computing Sytem," The First International Symposium on Data, Privacy, and E-Commerce, page(s): 310-315, November 1-3, 2007.

[Keen05] Keeney, M., et al.: Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sector. Technical report, US Secret Service and CERT Program,SEI, CMU, Pittsburgh, PA (May 2005)

[Ko94] Calvin Ko, George Fink and Karl Levitt, "Automated Detection of Vulnerabilites in Privileged Programs by Execution Monitoring," 10th Annual Computer Security Applications Conference Proceedings, page(s): 134-144, Orlando, FL, December 5-9, 1994.

[Kole05] Oleg Kolesnikov and Wenke Lee. "Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic." Technical Report GIT-CC-05-09, Georgia Institute of Technology, 2005.

[Kos97] Andrew P. Kosoresow, Steven A. Hofmeyr, "Intrusion Detection via System Call Traces," IEEE Software, v.14 n.5, p.35-42, September 1997.

[Lar99] James Larus, 'Whole Program Paths", Proceedings of the SIGPLAN 1999 Conference on Programming Languages Design and Implementation, Atlanta, GA. May 1999.

المنارة للاستشارات
www.manaraa.com

[Levi95] John Levine, Tony Mason, Doug Brown, "lex & yacc", O'Reilly & Associates, Inc., 1995.

[Lipt77] R. J. Lipton and L. Snyder, "A Linear Time Algorithm for Deciding Subject Security", Journal of Association for Computer Machinery, Vol. 24, No 3, July 1977, pp. 455-464

[Losc05] Peter Loscocco, Stephen Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," SELinux Symposium, 2005

[Mann03] Scott Mann, Ellen Mitchell, Mitchell Krell, "Linux System Security: An Administrator's Guide to Open Source Security Tools", Prentice Hall, 2003.

[Mcca05] Bill McCarty, SELINUX NSA's Open Source Security Enhanced Linux, O'Reilly & Associates, 2005

[Mins67] Marvin Minsky, Computation: Finite and Infinite Machines, Prentice-Hall, Inc., 1967

[Mitc97] Tom Mitchell, Machine Learning, WCB/McGraw-Hill, 1997

[NIST1] http://niap.nist.gov/cc-scheme/cc_docs/index.html, National Institute of Standards and Technology

[NSA01] http://www.nsa.gov/selinux/index.cfm, National Security Agency, Central Security Service, Fort George G. Meade, MA, NSA Press Release, January 2001.

[Park03] Jaehong Park, Usage Control: A Unified Framework for Next Generation Access Control, PhD dissertation, George Mason University, Fairfax VA., Summer 2003

[RACF03] z/OSV1R6.0 Security Server RACF Security Administrator's Guide, IBM Corporation, SA22-7683-05, August 8, 2003

[Rahi04] Niki Rahimi *(IBM)*, "Trusted Path Execution for the Linux 2.6 Kernel as a Linux Security Module," 14th USENIX Security Symposium, pages 73–80 of the *Proceedings,* June 2004.

[Ravi04] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. "Security in Embedded Systems: Design Challenges." *ACM Transactions on Embedded Computing Systems*, 3(3), August 2004.

[Rich08] Elaine Rich, "Automata, Computability, and Complexity," Prentice Hall, Upper Saddle River, NJ, 2008.

[Rose99] Kenneth Rosen, "Discrete Mathematics and Its Applications," 4th Edition, WCB/McGraw-Hill, 1999

المنارة للاستشارات
www.manaraa.com

[SCC70] Security Controls for Computer Systems: Report of Defense Science Board Task Force on Computer Security, Rand Corporation, Office of the Director of Defense Research and Engineering, Washington D.C., February, 1970.

[Schr74-1] Barry Schrager, SHARE VS/OS Security and Data Management Project Goals for Data Security, SHARE Conference, March 4, 1974.

[Schr74-2] Barry Schrager, Centralized Resource Control Information Facility, IBM Data Security Forum, Denver, Co., September 1974.

[Sedg02] Robert Sedgewick, Algorithms in C: Part 5 - Graph Algorithms, 3rd Edition, Addison-Wesley, 2002.

[Sek01] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors", Proceedings of the 2001 IEEE Symposium on Security and Privacy, Page(s): 144-155, 2001.

[SHA99] 1955: IBM Customers form the First Computer User Group. Computer World, May 5th 1999; SHARE is IBM user group organization established in 1955 (the first such organization). It met for the first time at Rand Corporation in Santa Monica, CA on August 15th, 1955. SHARE works with IBM to evaluate and develop requirements and enhancements to IBM products and to the computer industry at large.

[Silb05] Abraham Silberschatz, Peter Galvin, Greg Gagne, "Operating System Concepts," 7th Edition, Addison-Wesley, 2005.

[Sips06] Michael Sipser, "Introduction to the Theory of Computation." 2nd edition, Thomson Course Technology, Section 1.4: Nonregular Languages, pp. 77–83. Section 2.3: Non-context-free Languages, pp. 123–129, 2006.

[Ski90] S. Skiena, "Minimum Spanning Tree." §6.2 in Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 232-236, 1990.

[Smal01a] Stephen Smalley, Timothy Fraser, and Chris Vance, "Linux Security Modules: General Security Hooks for Linux", http://lsm.immunix.org/ September, 2001

[Smal01b] Stephen Smalley, Timothy Fraser, "A Security Policy Configuration for the Security-Enhanced Linux", http://www.nsa.gov/selinux/papers/policy/policy.html, February 2001

[Smal04] Stephen Smalley, Chris Vance, Wayne Salamon, "Implementing SELinux as a Linux Security Module," Contract MDA904-01-C-0926, March 2004

[Spen99] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Anderson, Jay Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," Proceedings of The Eighth USENIX Security Symposium, August 1999, pages 123-139.

المنارة للاستشارات
www.manaraa.com

[Syma05] Symantec Critical System Protection A Technical White Paper given to me by Symantec representative, Summer 2005 (after PlatformLogic acquisition)

[Syma10] Symantec Global Internet Security Threat Report: Trends for 2009 A Technical White Paper, Volume XV, April 2010.

[Stall92] William Stallings, "Operating Systems", Macmillan Publishing Company, 1992.

[Tayl98] R. Gregory Taylor, Models of Computation and Formal Languages, Oxford University Press, 1998

[Wag01] D. Wagner and R. Dean, "Intrusion detection via static analysis", Proceedings of the 2001 IEEE Symposium on Security and Privacy, Page(s): 156-168, 2001.

[War99] C. Warrender, S. Forrest, B. Pearlmutter, "Detecting intrusions using system calls: alternative data models", Proceedings of the 1999 IEEE Symposium on Security and Privacy, Page(s):133 – 145, 1999.

[Wats95] Bruce W. Watson, A Taxonomy of Finite Automata Minimization Algorithms, Computer Science report 93/44, Eindhoven University of Technology, Netherlands, January 24, 1995.

[Wats00] Bruce W. Watson, Directly Constructing Minimal DFAs : Combining Two Algorithms by Brzozowski, in S. Yu and A. Paun, eds, CIAA 2000, London, Ontario, *Lecture Notes in Computer Science*, 2088(2001), 311–317, Springer, 2000.

[Webb08] Charles F. Webb, IBM, z10: The Next-Generation Mainframe Microprocessor, *IEEE Micro*, vol. 28, no. 2, pp. 19-29, March/April, 2008.

[Wrig02a] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman, "Linux Security Module Framework," 11th USENIX Security Symposium, August 2002

[Wrig02b] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," 11th USENIX Security Symposium, August 2002.

المنارة للاستشارات
www.manaraa.com

Appendix A: Prototype Source Code

A.1 PPTM Source Code

```
/*********************************************************************************
/*  PPTM - Program Pathing Trust Model
/*********************************************************************************
/*
/*  Purpose of this program is to simulate an automata for the purposes of mapping
/*  a sequence of process invocation calls.   The program will then validate those
/*  calls using the built automata.
/*
/*  Developer: Robert Dahlberg - PhD candidate
/*            Virginia Commonwealth University
/*            Computer Science Department
/*            School of Engineering
/*            Prototype as partial fulfillment of PhD dissertation
/*
/*  Created:     Febuarary 7th, 2010
/*  Updated:     January 23rd, 2011
/********************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Automata.h"
#include "print.h"
/**************************************************
/*            define global variables
/**************************************************/
acell * scnatm(anchor * ack, char * pgm);
void traceaut(anchor * ank);
const char blank[] = " ";
char * bid = NULL;
char * bcaller = NULL;
char * bcalled = NULL;
char * vid = NULL;
char * vcaller = NULL;
char * vcalled = NULL;
char * last = NULL;
anchor   * achr;
stack    * stk;
automata * autom;
char TFile[200] = "train";
char tbuffer[80] = " ";
char VFile[200] = "validate";
char vbuffer[80] = " ";
char AFile[200] = "auditfile";
char abuffer[80] = " ";


main() {
```


www.manaraa.com

```
/*********************************************/
/*  allocate storage for anchor data area    */
/*********************************************/
achr = (anchor *) malloc(sizeof(anchor));
achr->len = sizeof(anchor);
strcpy (achr->eyecat,"*ANCHOR*");
achr->stkptr = 0;
achr->sptr = 0;
achr->snum = 0;
achr->autptr = 0;
achr->aptr = 0;
achr->anum = 0;
printf ("  anchor pointer = %p \n",achr);

/*********************************************/
/*   allocate storage for Stack data area    */
/*********************************************/
stk = (stack *) malloc(sizeof(stack));
memset(stk,'\0',sizeof(stack));
stk->len = sizeof(stack);
strcpy (stk->eyecat, "**STACK*");
stk->nxtstk = 0;
achr->stkptr = stk;
achr->sptr = & stk->cellstk[0];
achr->snum = 0;
printf ("   stack pointer = %p\n", stk);

/***********************************************/
/*  allocate storage for Automata data area    */
/***********************************************/
autom = (automata *) malloc(sizeof(automata));
memset(autom,'\0',sizeof(automata));
autom->len = sizeof(automata);
strcpy (autom->eyecat, "AUTOMATA");
achr->autptr= autom;
achr->aptr = & autom->autcell[0];
achr->anum = 0;
printf ("automata pointer = %p\n", autom);

/************************************************/
/*    Build Automata from recorded system       */
/************************************************/
/************************************************/
/*            Open Training file                */
/************************************************/
acell * aelement = NULL;
acell * a3element = NULL;
scell * aselement = NULL;
acell * a2element = NULL;
scell * lselement = NULL;
scell * selement = NULL;

FILE *tfp;
  if((tfp = fopen(TFile,"r")) == NULL)
  {
    printf("Cannot OPEN " BOLDBLACK "train" RESET " file \n");
```


المنارة للاستشارات
www.manaraa.com

```
    exit(1);
  }

  /********************************************
  /*          read training record          */
  /********************************************/
while(fgets(tbuffer,sizeof(tbuffer),tfp)!= NULL)
{
  bid = strtok(tbuffer,blank);
  printf("ID = %s \n",bid);
  bcaller = strtok(NULL,blank);
  printf("Caller = [%s] \n",bcaller);
  bcalled = strtok(NULL,blank);
  printf("Called = [%s] \n",bcalled);

/***********************************************/
/*      Find caller process in automata       */
/***********************************************/
  aelement = scnatm(achr,bcaller);      /* find caller's acell in automata      */
/**********************************************/
/**********************************************/
/*      "caller" not found in automata       */
/**********************************************/
/**********************************************/
  if (aelement == NULL)                 /* caller's acell not found             */
    {

    /**********************************************
    /*          Add "caller" to Automata          */
    /**********************************************/

    aelement = achr->aptr;              /* get new acell                      */
    strcpy(aelement->pgm,bcaller);      /* copy caller process into new acell   */
    achr->anum = achr->anum + 1;        /* update acell number by one           */
    achr->aptr = &autom->autcell[achr->anum]; /* ptr to next available  acell    */

    selement = achr->sptr;              /* get next available unused scell      */
    stk = achr->stkptr;                 /* get pointer to stack                 */
    achr->snum = achr->snum + 1;        /* update scell number by one           */
    achr->sptr = &stk->cellstk[achr->snum]; /* get next available unused scell   */
    aelement->lnkcell = selement;       /* store 1st available scell in new acell*/
      /******************************************************/
      /* printf("Caller = [%s] not found\n",bcaller);
      /******************************************************/
    a2element = scnatm(achr,bcalled);   /* find calling process acell in automata*/
    /**********************************************/
    /*     "called" process not in automata       */
    /**********************************************/
    if (a2element == NULL)              /* if caller process not found          */
      {
        a2element = achr->aptr;          /* get an acell for calling process    */
      strcpy(a2element->pgm,bcalled);    /* move calling process into acell      */
      a2element->lnkcell = NULL;         /* clear acell's link to scell          */
      achr->anum = achr->anum + 1;       /* update scell number by one           */
      achr->aptr = &autom->autcell[achr->anum];  /* point to next free acell     */
      }
```


المنارة للاستشارات
www.manaraa.com

```
 selement->pgmcell = a2element;        /* move called acell to scell of caller */
 selement->lnkcell = NULL;             /* clear scell's next scell pointer    */
}
else
{
/**********************************************/
/**********************************************/
/*            "Caller" process found           */
/**********************************************/
/**********************************************/
   a2element = NULL;                   /* clear a2element            */
 a2element = scnatm(achr,bcalled);     /* find if an acell for called */
/************************************************************/
/*  called process NOT found - make sure called acell in scell */
/************************************************************/
 if (a2element == NULL)
   {
    selement = aelement->lnkcell;     /* get 1st scell out of acell    */
    if (selement == NULL)
      {
       stk = achr->stkptr;            /* get stack */
       aelement->lnkcell = achr->sptr; /* point last scell to new scell */
       selement = achr->sptr;          /* get next free scell  */

       achr->snum = achr->snum + 1;    /* increment scell number by one */
       achr->sptr = &stk->cellstk[achr->snum]; /* advance next free scell ptr*/

       a2element = achr->aptr;         /* get next free acell ptr  */
       achr->anum = achr->anum + 1;    /* increment acell number by one  */
       achr->aptr = &autom->autcell[achr->anum]; /* advance to next acell ptr*/

       strcpy(a2element->pgm,bcalled); /* copy called process name to new acell */

       selement->pgmcell = a2element;  /* point to new acell from new scell */
       selement->lnkcell = NULL;       /* init new scell pointer to next scell*/
      }

     else
      {
       a3element = selement->pgmcell;  /* get acell out of 1st scell    */
       lselement = selement;
       /***************************************************************/
       /*  search scells in found "called" acell for "caller" process acell  */
       /***************************************************************/
       while ((a3element != a2element) && (selement != NULL))
         {
          lselement = selement;        /* save this scell as last scell */
          selement = lselement->lnkcell;
          if (selement != NULL)
            {
             a3element = selement->pgmcell; /* get acell out of next scell  */
            }
         }
        /************************************
                        /*        was NO scell found?
                        /**********************************/
```


المنارة للاستشارات
www.manaraa.com

```
                          if (selement == NULL)
                                  {
           if (a3element != a2element)
             {
              stk = achr->stkptr;              /* get stack */
              lselement->lnkcell = achr->sptr; /* point last scell to new scell */
              selement = achr->sptr;           /* get next free scell  */

              achr->snum = achr->snum + 1;     /* increment scell number by one */
              achr->sptr = &stk->cellstk[achr->snum]; /* advance next scell ptr*/
              a2element = achr->aptr;          /* get next free acell ptr  */
              strcpy(a2element->pgm,bcalled); /* copy called process name to acell */

              selement->pgmcell = a2element;  /* point to new acell from new scell */
              selement->lnkcell = NULL;       /* init new scell ptr to next scell*/
              achr->anum = achr->anum + 1;    /* increment acell number by one  */
              achr->aptr = &autom->autcell[achr->anum]; /* advance to next acell*/
             }
          }
                 }
}
   else
{
/**************************************************************/
/*  called process FOUND     - make sure called acell in scell */
/**************************************************************/
selement = aelement->lnkcell;
   if (selement == NULL)
   {
    stk = achr->stkptr;              /* get stack */
    aelement->lnkcell = achr->sptr; /* point last scell to new scell */
    selement = achr->sptr;           /* get next free scell  */

    achr->snum = achr->snum + 1;     /* increment scell number by one */
    achr->sptr = &stk->cellstk[achr->snum]; /* advance next free scell ptr*/

    selement->pgmcell = a2element;  /* point to new acell from new scell */
    selement->lnkcell = NULL;       /* init new scell pointer to next scell*/
   }
     else
            {
    while ((a3element != a2element) && (selement != NULL))
       {
       lselement = selement;            /* save this scell as last scell */
       selement = lselement->lnkcell;
       if (selement != NULL)
          {
           a3element = selement->pgmcell;  /* get acell out of next scell  */
          }
       }
    if (a3element != a2element)
       {
       stk = achr->stkptr;              /* get stack */
       lselement->lnkcell = achr->sptr; /* point last scell to new scell */
       selement = achr->sptr;           /* get next free scell  */
```


المنارة للاستشارات
www.manaraa.com

```
              achr->snum = achr->snum + 1;     /* increment scell number by one */
              achr->sptr = &stk->cellstk[achr->snum]; /* advance to next scell ptr*/

              selement->pgmcell = a2element;  /* point to new acell from new scell */
              selement->lnkcell = NULL;       /* init new scell ptr to next scell*/
             }
          }
       }
    }
memset(tbuffer,'\0',sizeof(tbuffer));
}
 fclose(tfp);
 traceaut(achr);
/***************************************************/
/*            Open Validate file              */
/***************************************************/
printf("\nProcessing Validate file \n\n");
FILE *vfp;
if((vfp = fopen(VFile,"r")) == NULL) {
  printf("Cannot OPEN " BOLDBLACK "validate" RESET " file \n");
  exit(1);  }

/***************************************************/
/*            Open auditfille file              */
/***************************************************/
printf("\nProcessing auditfile file \n\n");
FILE *afp;
if((afp = fopen(AFile,"w")) == NULL)
       {
     printf("Cannot OPEN " BOLDBLACK "auditfile" RESET " file \n");
    exit(1);
       }

/***************************************************/
/*         Validate a Process                 */
/***************************************************/
while(fgets(vbuffer,sizeof(vbuffer),vfp)!= NULL)
{
  vid = strtok(vbuffer,blank);
  printf("ID = [%s] \n",vid);
  vcaller = strtok(NULL,blank);
  printf("Caller = [%s] \n",vcaller);
  vcalled = strtok(NULL,blank);
  printf("Called = [%s] \n",vcalled);
  /***************************************************
  /*     Validate Process invocation calls
  /***************************************************/
  memset(abuffer,'\0',sizeof(abuffer));
  aelement = scnatm(achr,vcaller);       /* find caller acell in automata       */
   if (aelement == NULL)
    {
      strcpy(abuffer,"Caller process [");
      strcat(abuffer,vid);
      strcat(abuffer,"] - [");
      strcat(abuffer,vcaller);
      strcat(abuffer,"] invalid \n");
```


المنارة للاستشارات
www.manaraa.com

```
    fputs(abuffer,afp);
    printf("Caller process %s - [%s] invalid \n",vid,vcaller);
  }
else
  {
    a2element = scnatm(achr,vcalled);      /* find called acell in automata  */
    if (a2element == NULL)
    {
      strcpy(abuffer,"Called process [");
      strcat(abuffer,vid);
      strcat(abuffer,"] - [");
      strcat(abuffer,vcalled);
      strcat(abuffer,"] invalid \n");
      fputs(abuffer,afp);
      printf("Called process %s - [%s] invalid \n",vid,vcalled);
    }
    else
     {
      selement = aelement->lnkcell;
                 if (selement != NULL)
                   {
        a3element = selement->pgmcell;
                   while ((a3element != a2element) && (selement != NULL))
          {
           lselement = selement;            /* save this scell as last scell */
           selement = lselement->lnkcell;
           if (selement != NULL)
             {
              a3element = selement->pgmcell;  /* get acell out of next scell  */
             }
          }
        if (a3element != a2element)
          {
           strcpy(abuffer,"It is invalid for PID [");
           strcat(abuffer,vid);
           strcat(abuffer,"] - process [");
           strcat(abuffer,vcaller);
           strcat(abuffer,"] to call process [");
           strcat(abuffer,vcalled);
           strcat(abuffer,"] \n");
           fputs(abuffer,afp);
           printf("It is invalid for process %s - [%s] to call process [%s] \n",vid,vcaller,vcalled);
          }
       }
                 else
                   {
                          strcpy(abuffer,"It is invalid for PID [");
        strcat(abuffer,vid);
        strcat(abuffer,"] - process [");
        strcat(abuffer,vcaller);
        strcat(abuffer,"] to call process [");
        strcat(abuffer,vcalled);
        strcat(abuffer,"] \n");
        fputs(abuffer,afp);
        printf("It is invalid for process %s - [%s] to call process [%s] \n",vid,vcaller,vcalled);
                          }
```


المنارة للاستشارات
www.manaraa.com

```
        }
       }
memset(abuffer,'\0',sizeof(abuffer));
}
fclose(vfp);
fclose(afp);

printf(BOLDBLACK"Exit Program" RESET "\n");

return 0;
}

/*****************************************************/
/****************************************************/
/* Subroutine:  scnatm                          */
/*-------------------------------------------------*/
/*       search automata for a program name         */
/****************************************************/
/****************************************************/
acell * scnatm(anchor * ack, char * pgm)
{
 automata * atm = ack->autptr;
 int xno = ack->anum;
 int i = 0;
 int finda = 0;
 acell * xelement = NULL;

 xelement = &atm->autcell[i];
 /*************************************************************************/
 /* Scan Automata until end of automata sting or found the process name  */
 /*************************************************************************/
 while ((xno != i) && (finda == 0))
   {
    /*******************************************/
    /**   Matching program found in automata  **/
    /*******************************************/
    if (strcmp(xelement->pgm,pgm) == 0)     /* compare acell processes */
      {
       finda = 1;                         /* found it - mark flag    */
      }
    /*******************************************/
    /**   Get next process name in automata   **/
    /*******************************************/
    else
      {
       i++;                              /* advance to next acell in automata */
       xelement = &atm->autcell[i];           /* get @ of next acell in automata */
      }
   }
 /*******************************************/
 /*      Was process name not found?        */
 /*******************************************/
    if (finda == 0)                     /* no match found - mark return element */
   {
    xelement = NULL;                    /* mark return element to NULL      */
   }
```


المنارة للاستشارات
www.manaraa.com

```
 return xelement;
}

/*******************************************************/
/*******************************************************/
/* Subroutine:  traceaut                              */
/*---------------------------------------------------*/
/*    Trace automata and print out programs             */
/*******************************************************/
/*******************************************************/
void traceaut(anchor * ank)
{
 char TraceFile[200] = "automtrace";
 char Tracebuffer[200] = " ";
 scell * scelement = NULL;
 acell * aaelement = NULL;
 acell * abelement = NULL;
 automata * ama = ank->autptr;
 int ano = ank->snum;
 int sno = ank->anum;
 int j = 0;
 int x =0;
 char cella[24];
 char cells[24];

FILE *tracefp;
if((tracefp = fopen(TraceFile,"w")) == NULL)
    {
     printf("Cannot OPEN " BOLDBLACK "autotrace" RESET " file \n");
     exit(1);
       }

 printf("\n Trace Automata \n");
 printf("number of acells: [%d] \n",sno);

 strcpy(Tracebuffer,"                Automata Trace \n");
 fputs(Tracebuffer,tracefp);
 strcpy(Tracebuffer,"-------------------------------------------------------\n");
 fputs(Tracebuffer,tracefp);
 memset(Tracebuffer,'\0',sizeof(Tracebuffer));
 strcpy(Tracebuffer,"Caller Process -> Called Process |Called Process \n");
 fputs(Tracebuffer,tracefp);
 strcpy(Tracebuffer,"-------------------------------------------------------\n");
 fputs(Tracebuffer,tracefp);
 memset(Tracebuffer,'\0',sizeof(Tracebuffer));
 aaelement = &ama->autcell[j];
 /*********************************************************************/
 /* Scan Automata until end of automata sting or found the process name  */
 /*********************************************************************/
 while (j < sno)
    {
    /********************************************/
    /**   Matching program found in automata  **/
    /********************************************/
    printf("Calling = [%s]",aaelement->pgm);
    strcpy(Tracebuffer,aaelement->pgm);
```


www.manaraa.com

```
    scelement = aaelement->lnkcell;

    /******************************************/
    /**   Get next process name in automata   **/
    /******************************************/
    while ((scelement != NULL))
      {
       abelement = scelement->pgmcell;
                 if (x == 0)
                   {
                        printf("->[%s]",abelement->pgm);
                        strcat(Tracebuffer,"->");
                        x ++;
                       }
                else
                       {
                        printf("|[%s]",abelement->pgm);
                        strcat(Tracebuffer,"|");
                        if (strlen(Tracebuffer) >= (200-9))
                                {
                                 printf("\n");
                                 strcat(Tracebuffer,"\n");
                                 fputs(Tracebuffer,tracefp);
             memset(Tracebuffer,'\0',sizeof(Tracebuffer));
                                }
                       }
       strcat(Tracebuffer,abelement->pgm);
       scelement = scelement->lnkcell;
      }
     if (strlen(Tracebuffer) >= 2)
        {
       printf("\n");
       strcat(Tracebuffer,"\n");
       fputs(Tracebuffer,tracefp);
                 memset(Tracebuffer,'\0',sizeof(Tracebuffer));
                 }
    j++;
      x = 0;
    aaelement = &autom->autcell[j];
   }
 printf("close auditfile \n");
 snprintf(cella, sizeof(cella), "%d", ank->anum - 1);
 strcpy(Tracebuffer,"Number of acells =");
 strcat(Tracebuffer,cella);
 strcat(Tracebuffer,"\n");
 fputs(Tracebuffer,tracefp);
 memset(Tracebuffer,'\0',sizeof(Tracebuffer));
 snprintf(cells, sizeof(cells), "%d", ank->snum - 1);
 strcpy(Tracebuffer,"Number of scells =");
 strcat(Tracebuffer,cells);
 fputs(Tracebuffer,tracefp);
 fclose(tracefp);
 return;
}
```


المنارة للاستشارات
www.manaraa.com

A.2 PPTM Automata Header files

```
#define n 1000000
#define m 100000

/****************************************/
/*  link list scells
/****************************************/
struct scell
       {
        struct acell         *          pgmcell;
        struct scell         *          lnkcell;
       };
/*****************************************
/*  Stack of link list scells          */
/****************************************/
struct  stack
       {
        int                                      len;
        char                                     eyecat[8];
        struct stack         *          nxtstk;
        struct scell                    cellstk[n];
       };
/*******************************************
/*  Automata acell                      */
/******************************************/
struct  acell
       {
        char      pgm[24];
        void      *          lnkcell;
       };
/*******************************************
/*  Audit pcell                        */
/******************************************/
struct pcell
       {
       struct acell  *  aptr;
       short int        acount;
       short int        status;
       struct pcell  *  dauptr;
       struct pcell  *  sibprt;
    };
/******************************************/
/*    Automata structure
/******************************************/
struct  automata
       {
        int                             len;
        char                            eyecat[8];
        struct acell         autcell[m];
       };
/*******************************************
/*   Process invocation sequence trace
/******************************************/
struct            IDtrace
```


المنارة للاستشارات
www.manaraa.com

```
  {
 int                                  len;
 char                    eyecat[8];
   struct pstack   * pstptr;    /* pstack pointer */
 struct pcell    * pptr;
 int              pnum;
    struct automata * invptr;   /* invalid processes */
 struct acell    * iaptr;
    int              ianum;
    struct automata * trctab;   /* Trace table */
 struct acell    * trcptr;
 int              trcnum;
 };
/******************************************
/*   Process stack
/******************************************/
struct           pstack
  {
 int                                        len;
 char                                       eyecat[8];
    struct pstack   *   nxtpstk;    /* next pstack pointer */
 struct pcell            pcellptr[n];
 };
/******************************************
/*    typedefs
/******************************************/
typedef struct acell acell;
typedef struct scell scell;
typedef struct stack stack;
typedef struct automata automata;
typedef struct IDtrace IDtrace;
typedef struct pcell pcell;
typedef struct pstack pstack;
/*******************************************
/*    System Anchor - main data area
/*******************************************/
typedef struct
  {
    int                  len;
    char                 eyecat[8];
    stack    *           stkptr;
    scell       *        sptr;
    int       snum;
    automata *           autptr;
    acell    * aptr;
    int                  anum;
 IDtrace  *  IDptr;
  }  anchor;
```

A.3 PPTM Print Header file

```
#define RESET       "\033[0m"        /* Reset Attribute */
#define BLACK       "\033[30m"        /* Black           */
```


المنارة للاستشارات
www.manaraa.com

```
#define RED         "\033[31m"         /* Red             */
#define GREEN       "\033[32m"         /* Green           */
#define YELLOW      "\033[33m"         /* Yellow          */
#define BLUE        "\033[34m"         /* Blue            */
#define MAGENTA     "\033[35m"         /* Magenta         */
#define CYAN        "\033[36m"         /* Cyan            */
#define WHITE       "\033[37m"         /* White           */
#define BOLDBLACK   "\033[1m\033[30m"  /* Bold Black      */
#define BOLDRED     "\033[1m\033[31m"  /* Bold Red        */
#define BOLDGREEN   "\033[1m\033[32m"  /* Bold Green      */
#define BOLDYELLOW  "\033[1m\033[33m"  /* Bold Yellow     */
#define BOLDBLUE    "\033[1m\033[34m"  /* Bold Blue       */
#define BOLDMAGENTA "\033[1m\033[35m"  /* Bold Magenta    */
#define BOLDCYAN    "\033[1m\033[36m"  /* Bold Cyan       */
#define BOLDWHITE   "\033[1m\033[37m"  /* Bold White      */
```


المنارة للاستشارات
www.manaraa.com

A.4 Testdata (automated data creation) Source Code

```
/*********************************************************************************
/*  Tstdata - generate test data for PPT
/*********************************************************************************
/*
/*  Purpose of this program is to generate test data to test the PPT program
/*
/*  Developer: Robert Dahlberg - PhD candidate
/*           Virginia Commonwealth University
/*           Computer Science Department
/*           School of Engineering
/*           Prototype as partial fulfillment of PhD dissertation
/*
/*           January 2th, 2011
/*
/*********************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "Test.h"
#include "print.h"
/**************************************************/
/*           define global variables            */
/**************************************************/

const char blank[] = " ";

struct ID
      {
        char ProID[3];
     char Pronum[21];
      };
char CallPro[24] = "        ";
 /*********************************************
 /*  CalledPro of link list scells          */
 /*********************************************/
struct  Pro
      {
        char     MID[1];
        char     MNum[23];
      };

char str[23];
char TFile[200] = "train";
char tbuffer[80] = " ";
int StrProc = 0;
int y = 0;
int x = 0;
int Process = 0;

main()
{
```


المنارة للاستشارات
www.manaraa.com

```
struct ID UID;
struct Pro CalledPro;
/**********************************************/
/*  OPEN Training file "train"              */
/**********************************************/
printf("Start Tstdata \n");
FILE *tfp;
if((tfp = fopen(TFile,"w")) == NULL)
     {
      printf("Cannot OPEN " BOLDBLACK "Train" RESET "file \n");
      exit(1);
     }
printf("OPENed train file \n");
/*************************/
/* Initialize variables  */
/*************************/
x = 0;
strcpy(CallPro,"S");
srand(time(NULL));
/************************************************/
/* Dowhile more stings are required            */
/************************************************/
while (x < NoIDs)
     {
      x = x + 1;                                          /*  add one to UID count  */
      strcpy(UID.ProID,"UID");
      snprintf(UID.Pronum, sizeof(UID.Pronum), "%d", x);

      /********************************************************/
      /*    determine random number of processes in the  sting   */
      /********************************************************/
      StrProc = 1 + rand() % NoProString;
      y = 0;

      /*********************************************/
      /*    Dowile more processes needed in string    */
      /*********************************************/
      while(y < StrProc)
              {
               memset(tbuffer,'\0',sizeof(tbuffer));
               /*********************************************/
               /*       randomly determine a process ID        */
               /*********************************************/
               Process = 1 + rand() % NoProcess;
               /*********************************************/
               /*sprintf(str, "%d", num);                             */
               /* str now contains "3" sprintf() is like printf() but outputs to a string. */
               /*********************************************/
               strcpy(CalledPro.MID,"M");
               snprintf(CalledPro.MNum, sizeof(CalledPro.MNum), "%d", Process);
               /************************************************************/
               /*     Construct training record and write to training file      */
               /************************************************************/
               strcpy(tbuffer,UID.ProID);
               strcat(tbuffer,"  ");
               strcat(tbuffer,CallPro);
```


المنارة للاستشارات
www.manaraa.com

```
                  strcat(tbuffer,"  ");
                  strcat(tbuffer,CalledPro.MID);
                  strcat(tbuffer," \n");
                  printf("[%s]\n",tbuffer);
                  fputs(tbuffer,tfp);
                  y = y + 1;
                  strcpy(CallPro,CalledPro.MID);
                 }
      strcpy(CallPro,"S");
     }
 printf("End of Tstdata \n");
 fclose(tfp);
 return 0;
}
```

A.5 Test Header file

```
#define NoProcess 100000 /* Number of processes to chose from  */
#define NoIDs 100000      /* Number of Stings in test data      */
#define NoProString 100  /* Max Number of processes per string */
```


www.manaraa.com

Appendix B: Glossary

Abnormal Process	Any process that does not accomplish or support the system's intended function.
Abnormal Process Behavior	The result of executing abnormal processes or executing normal processes in an invalid invocation sequence. *Abnormal behavior* is the complement of *normal behavior*.
Invalid Process Invocation Sequence	The set of *invalid process invocation sequences* is defined as the complement of *valid process invocation sequences*.
Critical Application	An application that must not be interrupted.
Critical System	A server dedicated to run a critical application. Interrupting, delaying or halting these systems can have dire consequences.
External Process	Those processes that have not been intentionally installed by a system administrator.
Internal Process	Those processes that have been intentionally installed on a computer system by a system administrator.
Invocation	When it is stated that, 'P1 invokes P2' it means that the CPU has executed an instruction from P1 and that the executed instruction has the intent of requesting the scheduler to place process P2 on the dispatch queue awaiting the CPU to execute P2's instructions
Normal Process	An internal or external process that conforms to the intended design specifications and/or supports the system's intended function
Normal Process Behavior	*Normal system behavior* is the result of executing only normal processes in a valid invocation sequence that supports a system's intended function.
Valid Process Invocation Sequence	*Valid process invocation sequences* are exactly those process invocation sequences that invoke a set of *normal processes* in a sequence that accomplishes or supports the system's intended primary function.
Process	A *process* is a program that is loaded into main memory and executed.
Process Invocation Sequence	A computer system does not just run a single process, but a sequence of processes. One process will invoke another, and so on. The execution of these processes should not occur in a random order. These processes should execute in a predetermined order.
Program	A *program* is a set of machine instructions that are organized in a logical sequence to perform a task or process.
System Integrity	An attribute of a system maintained to execute only *normal processes* in *valid process invocation sequences*.

المنارة للاستشارات
www.manaraa.com

Appendix C: ACF2® Program Pathing Defintion Module

No longer used by ACF2, ACF99@RB was a compiled program that provided a static dictionary of process invocation sequences that were authorized to gain access to resources. [ACF99]

```
         MACRO                                                   ACF 310 00010000
         ACF99@RB                                                        00020000
ACF99@RB TITLE 'ACF2 STRUCTURE MODEL MODULE'                     ACF 310 00030000
         PRINT ON,GEN,DATA             PRINT EVERYTHING          ACF 310 00040000
         COPY  ACFDOC                                            ACF 22  00050000
********************************************************************** 00060000
*                                                                    * 00070000
*                                                            * TS89408 00070100
* CHANGE LOG:                                                * TS89408 00070200
*                                                            * TS89408 00070300
*        THIS MODULE DEFINES TO ACF2 THE STRUCTURAL                  * 00080000
*        MODELS OF TSO COMMANDS AND MODULES TO ALLOW                 * 00090000
*        FOR MACRO DEFINITION OF PATH CONTROL.                       * 00100000
*                                                                    * 00110000
*  TK52778  05/22/89  REL 5.2                                    TK52778 00115000
*        NEW RELEASE OF MSPF VERSION 2.5                         TK52778 00115500
*                                                                        00115600
*  TK52021  09/27/89  REL 5.2                                    TK52021 00115700
*        COMMENT CARDS WITHOUT SEQUENCE NUMBERS                  TK52021 00115800
*                                                                        00115900
********************************************************************** 00120000
         EJECT                                                           00120100
************************   REL 6.0   ************************* TS89408 00120200
*                                                            * TS89408 00120300
*  TS88952  06/26/90                                         * TS88952 00120400
*        INUSRE PROGRAM PATHING GET CORRECT PROGRAM NAME     * TS88952 00120500
*        WHEN 'REXX' IS IN CONTROL.                          * TS88952 00120600
*                                                            * TS88952 00120700
*  TS89408  06/27/90                                         * TS89408 00120800
*        ALLOW PROGRAM PATHING FOR PROGRAMS CALLED FROM TSO  * TS89408 00120900
*        READY MODE.                                         * TS89408 00121000
*                                                            * TS89408 00121100
*  TS89418  06/27/90                                         * TS89418 00121200
*        FOR ISPTASK IN ISPF, SET THE ACTIVE PROGRAM TO BE   * TS89418 00121300
*        EITHER THE CURRENT PROGRAM, OR THE FIRST NON-APF    * TS89418 00121400
*        PROGRAM, NOT THE PROGRAM TO WHICH ISPTASK PASSED    * TS89418 00121500
*        CONTROL.                                            * TS89418 00121600
*                                                            * TS89418 00121700
*  TS89429  06/27/90                                         * TS89429 00121800
*        CORRECT IPCS COMMAND STRUCTURE FOR ESA 3.1.         * TS89429 00121900
*        ALLOW BLSUINI1 AND BLSQINI2.                        * TS89429 00122000
*                                                            * TS89429 00122100
*  TS89439  06/27/90                                         * TS89439 00122200
*        PREVENT INV-CMD EFFECT WITH TSO/E V2 USING CLISTS.  * TS89439 00122300
*        V2 BRANCH ENTERS A DEFINED MODULE AND DOESN'T CAUSE * TS89439 00122400
*        A PRB TO BE GENERATED.                              * TS89439 00122500
*                                                            * TS89439 00122600
*  TS90535  07/12/90                                         * TS90535 00122700
*        ALLOW THE 'EX' FORM OF TSO EXEC COMMAND TO ACCESS   * TS90535 00122800
*        CLIST LIBRARIES SET AS 'EXEC' FILES, AND NOT GET A  * TS90535 00122900
*        READ VIOLATION.                                     * TS90535 00123000
```


المنارة للاستشارات
www.manaraa.com

```
*                                                              * TS90535 00123100
*  TS90532 08/14/90                                            * TS90532 00123200
*        PREVENT 913 MESSAGES WHEN JCLCHECK IN PROCESS.        * TS90532 00123300
*        ALLOWS JCLCHECK TO VERIFY LOADLIBS AND NOT BE         * TS90532 00123400
*        FLAGGED BECAUSE IDCAMS IS CHECKING LOADLIBS.          * TS90532 00123500
*                                                              * TS90532 00123600
*  TS90878 07/15/91                                            * TS90878 00123800
*        VIOLATION OCCURRING AFTER TS91150 APPLIED.            * TS90878 00123900
*        PROGRAM ISRPCP APPEARS AS PROGRAM FOR VALIDATION      * TS90878 00124000
*                                                              * TS90535 00124100
*  TS91161  07/16/91                                           * TS91161 00124300
*        MODIFIY THE STRUCTURE PROCESSING TO GET THE PREVIOUS * TS91161 00124400
*        RB IF PL/I PROGRAM.                                   * TS91161 00124500
*                                                              * TS90535 00124600
*  TS91189  07/16/91                                           * TS91161 00124700
*        DEFINE 'EX' AS AN ALIAS FOR 'EXEC' FOR REXX.          * TS91161 00124800
*        CHANGE WAS MADE ON THE @CMD FOR EXEC.                 * TS91161 00124900
*                                                              * TS90894 00125000
*  TS90894  07/22/91                                           * TS90894 00125100
*        DEFINE 'SASXA1' TCB, RB STRUCTURE FOR NEW SAS         * TS90894 00125200
*        RELEASE 6.06.                                         * TS90894 00125300
*                                                              * TS90894 00125500
*  TS90537  07/22/91                                           * TS90537 00125600
*        DEFINE JCLCHECK AND EDCHK TCB AND RB STRUCTURES.      * TS90537 00125700
*                                                              * TS90894 00125800
*  TS84746  10/11/91                                           * TS84746 00125900
*        ADD SUPPORT FOR SISTER TCB'S WHEN USING TO SUPPORT    * TS84746 00126000
*        APPLICATION MANAGER INTERFACE.                        * TS84746 00126100
*                                                              * TS95683 00126200
*  TS95683  04/03/92                                           * TS95683 00126300
*        ADD SUPPORT FOR TSPLUS REL 4.  DEFINE NEW STRUCTURE   * TS95683 00126400
*        'TSOSESS#' AND 'TSOSESS@'.                            * TS95683 00126500
*                                                              * TS95670 00126600
*  TS95670  04/03/92                                           * TS95670 00126700
*        ADD ENDEVOR COMMAND STRUCTURE SUPPORT.                * TS95670 00126800
*                                                              * TS95670 00126900
*  TS93164  09/02/92                                           * TS93164 00127000
*        DEFINE ISPICP UNDER ISPF.                             * TS93164 00127100
*                                                              * TS93164 00127200
*  TS95935  09/08/92                                           * TS95935 00127301
*        ADD SUPPORT FOR TSO/E PLATCMD.                        * TS95935 00127401
*                                                              * TS93164 00127503
*  TS98124  10/01/92                                           * TS98124 00127603
*        ADD SAS 6.06 AND 6.07 PGM SASHOST                     * TS98124 00127703
*                                                              * TS98124 00127803
*  TS95948  10/01/92                                           * TS95948 00127903
*        ADD SUPPORT FOR SPIFFY PRODUCT                        * TS95948 00128003
*                                                              * TS95948 00128103
*                                                              *         00128200
**************************************************************** TS89408 00129000
         EJECT                                                          00129010
************************   REL 6.2   **************************         00129012
*                                                              *         00129014
*  TA0378A  12/09/93                                           * TA0378A 00129016
*        ADD IKJEXC2 AS ALIAS OF EXEC                          * TA0378A 00129018
*                                                              *         00129020
*  TA1028C  06/17/94  Z0006                                    * TA1028C 00129022
*        ADD DB2'S DSN COMMAND STRUCTURE                       * TA1028C 00129024
*                                                              *         00129026
*  TA0946C  06/20/94  Z0009                                    * TA0946C 00129028
*        ADD SASXAL7                                           * TA0946C 00129030
*                                                              *         00129032
*  TA1389C  06/20/94  Z0009                                    * TA1389C 00129034
*        REMOVE #PLI,#JCLCHK,#EDCHK FROM 2ND SPF @TCB          * TA1389C 00129036
*        TO REMOVE PGM-PATH INV-CMD VIO FOR JCLCHECK           * TA1389C 00129038
*                                                              *         00129040
*  TA1792C  11/17/94  Z0008                                    * TA1792C 00129042
*        FIX TA1389, ADDED BACK #PLI,#JCLCHK,#EDCHK.           * TA1792C 00129044
*        SOURCED FIX: REMOVE #ISRPTC FROM SPF @TCB.            * TA1792C 00129046
*                                                              *         00129048
* END OF LOG.                                                  *         00129050
```


المنارة للاستشارات
www.manaraa.com

```
*****************************************************************         00129052
         SPACE 1                                                          00130000
ACF99@RB CSECT                                                   ACF 22   00140000
         SPACE 1                                                 ACF 22   00150000
*                                                                ACF 310 00160000
*                                                                ACF 310 00170000
*        SPF COMMAND STRUCTURE                                   ACF 310 00180000
*        ---------------------                                   ACF 310 00190000
*                                                                ACF 310 00200000
SPF      @CMD  ISPF,PDF,ISPSTART,MULTISPF,MSPF,ISRPCP,ISPICP     TS90878 00210000
*                                                                TS93164 00211000
         @TCB  (#SPF,#ISPF,#ISPSTAR,#PDF,#ISPICP,#ISRPCP,        TS74831X00220000
               #MSPF,#MULTSP),                                   TS74831X00221000
               (#ISPMAIN,#SPFMAIN)                               TS77767 00230000
         @TCB  (#ISPTASK,#TSOSESS),                      TS95683,TS77851X00240000
               (#MMAIN,#ISRYXDR,#ISPANRC,#IPNRECV,       TK52794,TA1389CX00251000
               #PLI,#JCLCHK,#EDCHK,                      TK52794,TA1389CX00251100
               #ISPXC,                   TS91161,TK52794,TA1389C,TA1792CX00251200
               #ISPXP),                  TS91161,TK52794,TA1389C,TA1792CX00251300
               FLAGS=SISTER                                      TS74452 00252000
#MSPF    @RB   MSPF,CMD=CMD,                                     TK52778X00253000
               NEXT=(RB,#MISPICP,#MISRPCP,#SPFMAIN,#ISPMAIN)     TK52778 00254000
#MULTSP  @RB   MULTISPF,CMD=CMD,                                 TK52778X00255000
               NEXT=(RB,#MISPICP,#MISRPCP,#SPFMAIN,#ISPMAIN)     TK52778 00255100
#MISPICP @RB   ISPICP,CMD=CMD,NEXT=(RB,#ISPMAIN,#SPFMAIN)        TK83561 00256000
#MISRPCP @RB   ISRPCP,CMD=CMD,NEXT=(RB,#ISPMAIN,#SPFMAIN)        TK83561 00257000
#SPF     @RB   SPF,CMD=CMD                                       TS77851 00260000
#SPFMAIN @RB     SPFMAIN,CMD=CMD                                 TS77851 00270000
#ISPF    @RB   ISPF,CMD=CMD                                      TS77851 00280000
#ISPMAIN @RB     ISPMAIN,CMD=CMD                                 TS77851 00290000
#ISPSTAR @RB   ISPSTART,CMD=CMD                                  TS77767 00300000
#PDF     @RB   PDF,CMD=CMD                                       TS77767 00310000
#ISPICP  @RB   ISPICP,CMD=CMD                                    TS77767 00320000
#ISRPCP  @RB   ISRPCP,CMD=CMD                                    TS77767 00330000
#ISRPTC  @RB   ISRPTC,CMD=(CDE,NEXTRB),END                       TK52794 00330100
*                                                                TS77851 00340000
#ISPTASK @RB   ISPTASK,LASTTCB,CMD=(CMD,NEXTTCB)                 TS77851 00350000
#TSOSESS @RB   TSOSESS*,LASTTCB,CMD=(CDE,NEXTTCB)                TS95683 00351000
#ISRYXDR @RB     ISRYXDR,NEXT=(RB,#ISRYXX),CMD=CMD               TS77851 00360000
#ISRYXX  @RB       ISRY**,NEXT=(RB,#ISPXC,#ISPXP),               TS77851X00370000
               LASTTCB,CMD=(CMD,NEXTTCB)                         TS77851 00380000
#ISPANRC @RB     ISPANREC,NEXT=(RB,#ISPXC)                       TS77851 00390000
#IPNRECV @RB     IPNRECV,NEXT=(RB,#ISPXC)                        TS51563 00391000
#JCLCHK  @RB     JCLCHECK,RENT,SYSLIB,END,CMD=CDE                TS90537 00392000
#EDCHK   @RB     EDCHECK,RENT,SYSLIB,END,CMD=CDE                 TS90537 00393000
#ISPXC   @RB   ********,LASTTCB,RENT,SYSLIB,CMD=(CDE,NEXTTCB),   TS89418X00400000
               NEXT=(RB,#JCLCHK,#EDCHK,#ISPXC,#ISPXP)    TS90537,TS89418 00401000
#ISPXP   @RB     ********,END,NORENT,NOSYSLIB,CMD=(CDE,NEXTTCB)  TS74452 00410000
*                                                                TS91161 00411000
*        FOR PL/I ASSIGN THE NAME OF THE CALLING PGM             TS91161 00412000
*                                                                TS91161 00413000
#PLI     @RB   IBMBOP**,RENT,SYSLIB,LASTTCB,NEXT=(RB,#ISPXC),    TS91161X00414000
               CMD=(CDE,PREVRB)                                  TS91161 00415000
*                                                                ACF 310 00420000
*                                                                TS74452 00430000
*        EXAMINE COMMAND STRUCTURE                               TS74452 00440000
*        --------------------                                    TS74452 00450000
*                                                                TS74452 00460000
#MMAIN   @RB   LTDMMAIN,NOSYSLIB,CMD=(CDE,NEXTTCB),LASTTCB,      TS74452X00510000
               NEXT=(RB,#M###0)                                  TS74452 00510100
#M###0   @RB   LTDM###0,NOSYSLIB,CMD=(CDE,NEXTTCB),LASTTCB,      TS74452X00510200
               NEXT=(RB,#MS##0)                                  TS74452 00510300
#MS##0   @RB   LTDM*##0,NOSYSLIB,CMD=(CDE,NEXTTCB),END           TS74452 00510400
*                                                                ACF 310 00560000
*                                                                ACF 310 00561000
*        XC COMMAND STRUCTURE                                    ACF 310 00570000
*        --------------------                                    ACF 310 00580000
*                                                                ACF 310 00590000
XC       @CMD  ,                                                 R41P166 00600000
         @TCB  #XC                                                       00610000
#XC      @RB   ********,NORENT,END,CMD=CDE                               00620000
```


www.manaraa.com

```
*                                                                      TS95670 00622000
*                                                                      TS95670 00623000
*        ENDEVOR COMMAND STRUCTURE                                     TS95670 00624000
*        -------------------------                                     TS95670 00625000
*                                                                      TS95670 00626000
NDVRC1   @CMD  ,                                                       TS95670 00627000
         @TCB  #NDVRC1                                                 TS95670 00628000
#NDVRC1  @RB   NDVRC1,CMD=(CMD,NEXTTCB),LASTTCB                        TS95670 00629000
*                                                                      TS95670 00629100
BC1PSRVL @CMD  ,                                                       TS95670 00629200
         @TCB  #BC1PSRV,(#ISPXC,#ISPXP)                                TS95670 00629300
#BC1PSRV @RB   BC1PSRVL,CMD=(CMD,NEXTTCB),LASTTCB                      TS95670 00629400
*                                                                      TS95670 00629500
BC1PSATT @CMD  ,                                                       TS95670 00629600
         @TCB  #BC1PSAT,(#ISPXC,#ISPXP)                                TS95670 00629700
#BC1PSAT @RB   BC1PSATT,CMD=(CMD,NEXTTCB),LASTTCB                      TS95670 00629800
*                                                                      ACF 310 00630000
*                                                                      TK86602 00631000
*        'PARALLEL TMP CALL' COMMAND STRUCTURE                         TK86602 00632000
*        -------------------------------------                         TK86602 00633000
*                                                                      TK86602 00634000
PTMPCALL @CMD  ,                                                       TK86602 00635000
         @TCB  (#EFF76,#PTCALL),NEXT=END                               TK86602 00636000
#EFF76   @RB   IKJEFF76,NEXT=(TCB,#FIBCMDS)                            TK86602 00637000
#PTCALL  @RB   ********,CALL,END,NORENT,NOSYSLIB,CMD=CDE               TK86602 00638000
*                                                                      TK86602 00639000
*                                                                      ACF 310 00640000
*        PARALLEL TMP FIB COMMANDS STRUCTURE                           TK86602 00641000
*        -----------------------------------                           TK86602 00642000
*                                                                      TK86602 00643000
#FIBCMDS @TCB  #EFF04                                                  TK86602 00644000
#EFF04   @RB   IKJEFF04,END,CMD='SUBMIT'                               TK86602 00645000
*                                                                      TK86602 00646000
*                                                                      TK86602 00647000
*        QED COMMAND STRUCTURE                                         ACF 310 00650000
*        ---------------------                                         ACF 310 00660000
*                                                                      ACF 310 00670000
QED      @CMD  Q                                                               00680000
         @TCB  (#QED,#Q)                                                       00690000
#QED     @RB   QED,END,CMD=(CMD,NEXTTCB)                                       00700000
#Q       @RB   Q,END,CMD=(CMD,NEXTTCB)                                         00710000
*                                                                      ACF 310 00720000
*                                                                      ACF 310 00730000
*        EDIT COMMAND STRUCTURE                                        ACF 310 00740000
*        ----------------------                                        ACF 310 00750000
*                                                                      ACF 310 00760000
EDIT     @CMD  E,IKJEBEMA,IKJEBECO                                             00770000
         @TCB  (#EDIT,#E1,#E2)                                         ACF 310 00780000
#EDIT    @RB   IKJEBE**,END,CMD=(CMD,NEXTTCB)                                  00790000
#E1      @RB   EDIT,END,CMD=(CMD,NEXTTCB)                              ACF 310 00800000
#E2      @RB   E,END,CMD=(CMD,NEXTTCB)                                 ACF 310 00810000
*                                                                      ACF 310 00820000
*                                                                      ACF 310 00830000
*        CALL COMMAND STRUCTURE                                        ACF 310 00840000
*        ----------------------                                        ACF 310 00850000
*                                                                      ACF 310 00860000
CALL     @CMD  SPFCALCP,IKJEFG00                                       TK86608 00870000
         @TCB  (#CALL,#SPFCALL,#KJEFG00,#TSOCALL,$TSOCALL),            TS89408X00880000
               FLAGS=SISTER                                            TS84746 00881000
#CALL    @RB   CALL,CALL,END,CMD=(CMD,NEXTTCB)                                 00890000
#KJEFG00 @RB   IKJEFG00,CALL,END,CMD=(CMD,NEXTTCB)                     TK86608 00890100
#SPFCALL @RB   SPFCALCP,CALL,END,CMD=(CMD,NEXTTCB)                             00900000
*   CALLED PGM FROM TSO READY MODE                                     TS89408 00900500
#TSOCALL @RB   ********,END,CMD=CDE                                    TS89408 00900600
$TSOCALL @RB   ********,END,NORENT,NOSYSLIB,CMD=CDE                    TS89408 00900700
*                                                                      ACF 310 00910000
*                                                                      ACF 310 00920000
*        ISPCALL COMMAND STRUCTURE                                     ACF 310 00930000
*        -------------------------                                     ACF 310 00940000
*                                                                      ACF 310 00950000
ISPCALL  @CMD  ,                                                       TS77106 00960000
```


www.manaraa.com

```
         @TCB  #ISPCALL                                          TS73712 00970000
#ISPCALL @RB   ISPCALL,CALL,END,CMD=(CMD,NEXTTCB)                TS73712 00980000
*                                                                ACF 310 00990000
*                                                                ACF 310 01000000
*        VSAPL COMMAND STRUCTURE                                 ACF 310 01010000
*        -----------------------                                 ACF 310 01020000
*                                                                ACF 310 01030000
*VSAPL   @CMD  ,                                                 R41P166 01040000
*        @TCB  #VSAPL                                            ACF 22  01050000
*        @TCB  #VSTAR,FLAGS=SISTER                               ACF 22  01060000
*#VSAPL  @RB   VSAPL,CMD=CMD                                     ACF 22  01070000
*#VSTAR  @RB   ASVPSTAR,NOSYSLIB,END,CMD=(CMD,NEXTTCB)           ACF 22  01080000
*                                                                ACF 310 01090000
*                                                                ACF 310 01100000
*        LIST COMMAND STRUCTURE                                  ACF 310 01110000
*        ----------------------                                  ACF 310 01120000
*                                                                ACF 310 01130000
LIST     @CMD  L,IKJEBLI1,IKJEBLI2,IKJEBLP1,IKJEBLM1 XL,XLIST    ACF 22  01140000
         @TCB  (#LIST,#L1,#L2)                                   ACF 310 01150000
#LIST    @RB   IKJEBL**,END,CMD=CMD                              ACF 22  01160000
#L1      @RB   L,END,CMD=CMD                                     ACF 310 01170000
#L2      @RB   LIST,END,CMD=CMD                                  ACF 310 01180000
*---------------------------------------------------------------*TS77534 01190000
* IPCS COMMAND STRUCTURE                                         TS77534 01200000
*---------------------------------------------------------------*TS77534 01210000
IPCS     @CMD  ,                                                 R41P166 01220000
         @TCB  #IPCS,#IPCSSUB                                    TS77534 01230000
#IPCSALL @TCB  #IPCSTSO                                          TS89429 01240000
#IPCS    @RB   IPCS,CMD=CMD                                      TS77534 01250000
#IPCSSUB @RB   BLS*****,CMD=CMD,NEXT=(TCB,#IPCSALL)              TS89429 01260000
#IPCSTSO @RB   BLS*****,CMD=(CMD,NEXTTCB),END                    TS89429 01270000
*                                                                TS89429 01270100
*                                                                TS79065 01271000
*                                                                TS79065 01271100
*        SAS COMMAND STRUCTURE                                   TS79065 01271200
*        ---------------------                                   TS79065 01271300
*                                                                TS79065 01271400
SASCP    @CMD                                                    TS79065 01271500
         @TCB  #SASCP                                            TS79065 01271600
         @TCB  (#SASLPA,#SAS,           TS98124,TS90894,TS79065,TA0946CX01271702
               #SASXA1,                 TS98124,TS90894,TS79065,TA0946CX01271704
               #SASHOST,#SASXAL7)       TS98124,TS90894,TS79065,TA0946C 01271706
         @TCB  (#SASCALL,#SASLIB)                                TS79065 01271800
#SASCP   @RB     SASCP,NOSYSLIB,CMD=CMD                  TS98124,TS79065 01271902
#SASLPA  @RB     SASLPA,NOSYSLIB,CMD=CMD                         TS79065 01272000
#SAS     @RB     SAS,END,CALL,NOSYSLIB,CMD=(CDE,NEXTTCB)         TS79065 01272100
#SASCALL @RB     SASCALL,END,CALL,NOSYSLIB,CMD=(CDE,NEXTTCB)     TS79065 01272200
#SASHOST @RB     SASHOST,END,NOSYSLIB,CMD=(CDE,NEXTTCB)          TS98124 01272302
#SASLIB  @RB     SASLIB,END,CALL,NOSYSLIB,CMD=(CDE,NEXTTCB)      TS79065 01272400
#SASXA1  @RB     SASXA1,END,CALL,NOSYSLIB,CMD=(CDE,NEXTTCB)      TS90894 01272500
#SASXAL7 @RB     SASXAL7,END,CALL,NOSYSLIB,CMD=(CDE,NEXTTCB)     TA0946C 01272530
*                                                                TA1028C 01273000
*                                                                TA1028C 01273010
*        DB2'S DSN COMMAND STRUCTURE                             TA1028C 01273020
*        ---------------------------                             TA1028C 01273030
*                                                                TA1028C 01273040
DSN      @CMD  ,                                                 TA1028C 01273050
         @TCB  #DSN                                              TA1028C 01273060
         @TCB  #ECP10,#DB2MASK                                   TA1028C 01273070
#DSN     @RB   DSN,CMD=CMD                                       TA1028C 01273080
#ECP10   @RB   DSNECP10,CMD=CMD                                  TA1028C 01273090
#DB2MASK @RB   ********,END,NORENT,NOSYSLIB,CMD=(CDE,NEXTTCB)    TA1028C 01273100
*                                                                TS88952 01275000
*                                                                TS88952 01275100
*        REXX COMMAND STRUCTURE                                  TS88952 01275200
*        ---------------------                                   TS88952 01275300
*                                                                TS88952 01275400
EXEC     @CMD  EX,IKJEXC2                                TA0378A TS91189 01275500
         @TCB    (#EXEC,#EX,#EXC2),FLAGS=SISTER          TA0378A TS89439 01275600
#EXEC    @RB     EXEC,END,CMD=(CDE,NEXTTCB)                      TS90535 01275700
#EX      @RB     EX,END,CMD=(CDE,NEXTTCB)                        TS90535 01275800
```


المنارة للاستشارات
www.manaraa.com

```
#EXC2    @RB     IKJEXC2,END,CMD=(CDE,NEXTTCB)          TA0378A TS90535 01275810
*                                                LINE DELETED BY TS89439 01275900
*                                                                TS95948 01276003
*        SPIFFY CMD                                              TS95948 01276103
*        ----------                                              TS95948 01276203
*                                                                TS95948 01276303
SPIFFY   @CMD  ,                                                 TS95948 01276403
         @TCB  (#SPIFFY)                                         TS95948 01276503
#SPIFFY  @RB   SPIFFY,LASTTCB,CMD=(CDE,NEXTTCB)                  TS95948 01276603
*                                                                TS95935 01276701
*                                                                TS95935 01277001
*        SUPPORT FOR TSO/E PLATCMD                               TS95935 01278001
*        -------------------------                               TS95935 01279001
*                                                                TS95935 01279101
IKJFCP03 @CMD  ,                                                 TS95935 01279201
         @TCB  #KJFCP03,FLAGS=SISTER                             TS95935 01279301
#KJFCP03 @RB   IKJFCP03,LASTTCB,RENT,SYSLIB,CMD=(CDE,NEXTTCB),   TS95935X01279401
               NEXT=(RB,#JCLCHK,#EDCHK,#PLI,#ISPXC,#ISPXP)       TS95935 01279501
         EJECT                                                   ACF 310 01280000
*                                                                ACF 310 01290000
*        COMMAND CROSS REFERENCE TABLE.                          ACF 310 01300000
*                                                                ACF 310 01310000
         @CXREF                                                          01320000
         SPACE 2                                                         01330000
         @ID  ,                                                          01340000
         SPACE 1                                                         01350000
         MEND                                                    ACF 310 01360000
         ACF99@RB                                                ACF 310 01370000
         END    ,                                                        01380000
```


المنارة للاستشارات
www.manaraa.com

Appendix D: Process Authentication

A method for authenticating that some other process is not masquerading as a process previously authorized to invoke another process, is essential to PPTM. This section outlines some possible to explore in future research for adding process authentication to the resulting PPTM solution model. This present research assumes all processes presented to the scheduler have been correctly authenticated.

An authentication verifies identity. Traditional authentication methods determine whether a user or resource is what it claims to be. Authentication of a user is traditionally determined by one or more factors such as *ownership*, *knowledge* or *inheritance* [Harr03]. *Ownership* usually translates into "something you have" such as a certificate, token, key or some such object that is uniquely issued to the user. *Knowledge* usually translates into "something you know" such as a password, the answers to a series of personal questions or an answer to a challenge. *Inheritance* usually translates into "something you are" such as a fingerprint or some other biometric signature; something that is physically unique the user. These factors were intended for user authentication and are not all appropriate to authenticate a process. For example, a process cannot "know" something and therefore cannot be authenticated by this factor.

Processes can be authenticated by *inheritance* and *ownership* factors. For instance, a process can have a certificate, thereby authenticated by "what it has." A process can also be authenticated by "what it is" using process characteristics, such as size, number of invocations or an associated hash value.

Additional authentication factors have emerged and are occasionally applied to users: *social networking*, *web-trust*, *location-based* and *time-based* [Harr03]. The authentication of processes

المنارة للاستشارات
www.manaraa.com

can use some of these factors, especially *location-based* and *time-based* factors as good indicators in authenticating processes.

In a process invocation sequence, each process must be authenticated to verify that the process is assigned the appropriate symbol from the alphabet Σ. The process name alone is not sufficient for authentication, since a process can masquerade as another process by using the same name. This is an area of the research that has not been addressed by other researchers. The three factors that would most likely best serve process authentication would be *ownership*, *inheritance* and *location-based*.

D.1 Ownership Authentication Factor

Process authentication could be effected using the *ownership* factor, if all processes had certificates as do some JAVA processes using JARS. However, this entails that all software development be required to start using digital certificates whenever a module is created, and this would be difficult to do.

D.2 Inheritance Authentication Factor

Using the *inheritance* factor in process authentication has potential. A digital hash such as SHA2, SHA1 or MD5 could be taken of a process at the time it is identified as a process and first scheduled for execution in a process invocation sequence. Then, whenever a process using the same name is encountered subsequently, a digital hash can be taken and compared to the hash taken when that named process was initially determined to be a *normal process* in a *valid invocation sequence*. If these two hashes match, then there is a very high likelihood that it is the same process. The only problem with this approach is that hash is CPU intensive and could

المنارة للاستشارات
www.manaraa.com

cause performance problems, although in some applications system criticality might justify the cost of additional hardware for this purpose.

The *inheritance* method can be found as a feature in Computer Associates CA Access Control® software. CA Access Control® [CAA08] creates hashes of all executables in the system and authenticates them before they execute. As maintenance is applied to these processes, new hashes are taken. Using CA Access Control® with the solution model described in this research could satisfy the authentication requirement.

D.3 Location-based Authentication Factor

The third method usable for process authentication is *location-based*. Identifying the directory from which a process is loaded is a good authentication method, if good access control is followed. If directories are well managed, then a directory from which a process is loaded is a good indication that the process being executed is the process intended. A process loaded from another directory would suggest that the process differs from the one intended to run. This is a preferred method, as it would not take much additional processing time to determine. Unfortunately, the Linux and UNIX OS do not save the name of the directory structure from which a process was loaded. The OS loader is independent of the OS scheduler. At the time the OS loads a file (for execution or otherwise) it does not know if the file is a data file or one that is used by the scheduler for execution. In the scheduler, the directory from which the process was loaded is not available in any of the data areas. Therefore, to authenticate a process with this method in these OS would require a modification of the OS kernel.

المنارة للاستشارات
www.manaraa.com

Appendix E: Is $\overline{L_v} \cup \overline{L(DFA_t)}$ = { }?

Because the DFA is built incrementally, it is not known at any time t whether $L(DFA_t) = L_v$. However, the strings in the set called the *white list* are in the non-empty intersection $L(DFA_t) \cap L_v$. Because the *white list* is also built incrementally it cannot be claimed that the set of strings called the *white list* is the set of stings $L(DFA_t)$. Furthermore, it is not known whether the set $L(DFA_t)$ merely forms a non-empty subset of L_v. $L(DFA_t)$ could contain a set of strings that are not a subset of L_v. That is, it is not known whether there exists another non-empty subset of strings both in $L(DFA_{t=5})$ and outside L_v. More formally, it is not known whether $\overline{L_v} \cup \overline{L(DFA_t)}$ = { }. This remains an open theoretical question and a topic for future research, as described in chapter 9.

The PPT model uses domain knowledge both to build DFA_t and to determine whether strings known to be in $L(DFA_t)$ are also in L_v. In this way the PPT model incrementally builds the non-empty intersection of $L(DFA_t)$ and L_v called the *white list*. However, the PPT DFA may also accept sequences that are invalid.

10.4 Other Approaches Making Assumptions similar to Is $\overline{L_v} \cup \overline{L(DFA_t)}$ = { }

Hofmeyr-Forrest [Hof98] and Ball-Larus [Ball92] assume that inferred strings are valid and recognizes this as an unproven assumption in later research [Ball96] [Laru99]. Hofmeyr-Forrest's n-gram approach used substrings of process invocation sequences to create patterns. Empirically they discovered that an n-gram of eleven processes was sufficient to discover anomalies in process invocation sequences. However, they did not validate whether the prefix n-gram preceding or the suffix n-gram following an n-gram were authorized.

المنارة للاستشارات
www.manaraa.com

Ball-Larus make the same assumption [Ball92]. They do not entertain the notion that the structure might infer paths that have not have been encountered. Ball-Larus use edge profiling to count the number of times a process path has been used. They hadn't identified 1) that they've made an assumption or 2) that once path between two processes is valid it is always valid regardless of whether or not there are prefix paths or suffix paths that were ever encountered. Larus discovers this assumption in later research [Laur99] and suggests that the whole process invocation sequence be validated. Over a series of articles he offers a number of solutions, such as process sequence probability and edge profiling.

E.2 Impact of the Assumption

One of the reasons why mapping *valid process invocation sequences* are so difficult to profile is due to all the possible invocation sequences that must be generated by a running a critical system as if it were in production. Each invocation sequence can have any number of variations, such as invoking processes for system or application services. These all produce multiple variations of an invocation sequence and must all be profiled.

It is almost impossible to profile all the possible valid invocation sequences due the complexity of all the various code paths in an application. Take for example an application that has error recovery processes that only gets invoked if an error occurs, or a process that only gets invoked if specific data is presented to the invoking process. These are process invocation sequences that are valid, but are not be profile-able in every case.

المنارة للاستشارات
www.manaraa.com

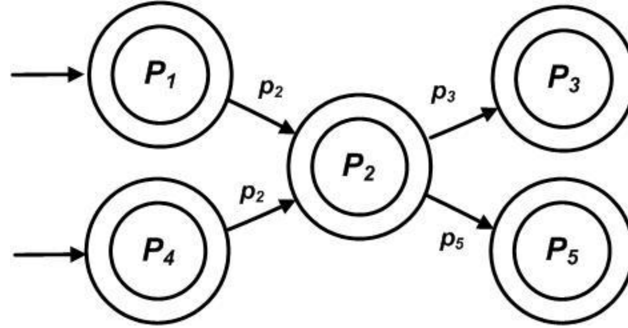

Figure E-1: Process Invocation Example

Using the potential DFA shown in figure E-1, assume that sequences $\boldsymbol{p_1p_2p_3}$ and $\boldsymbol{p_1p_2p_5}$ are *valid process invocation sequences* and that $\boldsymbol{P_2}$ invokes $\boldsymbol{P_5}$ only in rare occasions when an error occurs. The string $\boldsymbol{p_1p_2p_3}$ and $\boldsymbol{p_1p_2p_5}$ are both profiled by the PPT model because $\boldsymbol{P_2}$ was caused to fail. Also consider that $\boldsymbol{p_4p_2p_3}$ is profiled, but forcing $\boldsymbol{p_4p_2}$ to fail so it profiles the *valid process invocation sequence* $\boldsymbol{p_1p_2p_5}$ is difficult and therefore was not profiled because $\boldsymbol{P_2}$ could not be forced to fail when invoked by $\boldsymbol{P_4}$. Using domain knowledge, it is known that $\boldsymbol{p_1p_2p_5}$ is a *valid process invocation sequence*, but it was unsuccessfully profiled. In the PPT model assumption, this sequence is accepted by the language and considered a *valid process invocation sequence* because $\boldsymbol{p_4p_2p_5}$ is inferred. In this case the inferred process sequences, using domain knowledge, are valid. It is accepted by figure E-1, and if $\overline{\boldsymbol{L_v}} \cup \overline{\boldsymbol{L(DFA_t)}}$ proves true, all process invocation sequences accepted by PPTM are valid whether or not the sequences was previously encountered or inferred. But what if domain knowledge were to decide that $\boldsymbol{p_4p2p_5}$ is not valid? Because the answer to this question requires domain knowledge, it is unlikely a purely theoretical solution to the question would be satisfactory.

www.manaraa.com

Vita

Robert Andrew Dahlberg was born on May 27, 1954 in Alameda County, California at the U. S. Naval Hospital in Oakland, and is an American citizen. He graduated from Hononegah High School in Rockton, Illinois in 1972. He received a Bachelor of Arts from Western Illinois University, Macomb, Illinois in Philosophy and Comparative Religious Studies in 1976. He also received a Masters of Arts in Philosophy and a Master of Science in Computer Science, in 1982 from Northern Illinois University, DeKalb, Illinois. He subsequently became a Computer Science instructor in 1983 and taught 300 and 400 level courses in assembly language, PL/I, analysis and design, and JCL & Utilities for 4 semesters.

Bob has spent the majority of his career as a security professional specializing in access control, authentication, security designs and vulnerability assessment. He started his professional security career in 1984 at SKK, Inc., the creator of ACF2®, in Rosemont, Illinois where he served as a Security Software Developer and was first exposed to the concept of program pathing presented in this dissertation. SKK was later purchased by Computer Associates International in 1986 where Bob served as Manager of Software Development for ACF2®, Examine®, Pan-Audit®, and PRMS® for 8 years.

Bob became a Security Consultant to private industry in 1994 where he spent the next 9 years converting security system, developing security software solutions and assessing security systems. During that time, Bob started his own consulting company – Aware Computing Services, Inc. He worked as a consultant and developed security software designs for JME Software, Vasco Data Security International Inc., Computer Associates, EKC Inc., Blockade Systems Corporation, and Vanguard Integrity Professionals, Inc. Some companies Bob has consulted at are USB Swiss Bank, Sempra Energy, Ameritech, Cigna, TransUnion, Brown University, First National Bank of Chicago, NBD Bank, Wells Fargo and the Federal Reserve System.

Bob is currently full time employee at the Federal Reserve's National IT (FRIT) organization at the Federal Reserve of Richmond, where he presently serves as the National IT Design Review Board Chairman and Senior Technology Advisor. In his tenure at FRIT he also served as a Security Engineer for 5 years working on national IT applications and network infrastructure for the Federal Reserve.

Bob is a PhD student at Virginia Commonwealth University's School of Engineering in the Computer Science Department. Where in addition to his PhD work, he also works part time in Dr. Primeaux's Information Security Lab as a Senior Researcher. His research interest is primarily in the areas of Trusted Systems and Access Control, but is also extends to being actively involved in Computer Forensics for Apple's MAC and Security Design Methodologies.

المنارة للاستشارات
www.manaraa.com