2008-12

An application of Alloy to static analysis for secure information flow and verification of software systems

Shaffer, Alan B.

Monterey, California. Naval Postgraduate School, 2008.

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

AN APPLICATION OF ALLOY TO STATIC ANALYSIS FOR SECURE INFORMATION FLOW AND VERIFICATION OF SOFTWARE SYSTEMS

by

Alan B. Shaffer

December 2008

Dissertation Supervisor: Mikhail Auguston

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2008	**3. REPORT TYPE AND DATES COVERED** Doctoral Dissertation	
4. TITLE AND SUBTITLE: An Application of Alloy to Static Analysis for Secure Information Flow and Verification of Software Systems			**5. FUNDING NUMBERS**
6. AUTHOR(S) Alan B. Shaffer			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE** A

13. ABSTRACT ***(maximum 200 words)***

Within a multilevel secure (MLS) system, flaws in design and implementation can result in overt and covert channels, both of which may be exploited by malicious software to cause unauthorized information flows. To address this problem, the use of control dependency tracing has been explored to present a precise, formal definition for information flow. This work describes a security Domain Model (DM), designed in the Alloy formal specification language, for conducting static analysis of programs to identify illicit information flows, such as control dependency flaws and covert channel vulnerabilities. The model includes a formal definition for trusted subjects, which are granted extraordinary privileges to perform system operations that require relaxation of the mandatory access control (MAC) policy mechanisms imposed on normal subjects, but are trusted to behave benignly and not to degrade system security. The DM defines the concepts of program state, information flow and security policy rules, and specifies the behavior of a target program. The DM is compiled from a representation of the target program, written in a specialized Implementation Modeling Language (IML), and a specification of the security policy written in the Alloy language. The Alloy Analyzer tool is used to perform static analysis of the DM to detect potential security policy violations in the target program. This approach demonstrates that it is possible to establish a framework for formally representing a program implementation and for formalizing the security rules defined by a security policy, enabling the verification of that program representation for adherence to the security policy.

14. SUBJECT TERMS Security domain model, static analysis, automated program verification, trusted subjects, covert channels, dynamic slicing, specification language.			**15. NUMBER OF PAGES** 182
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN APPLICATION OF ALLOY TO STATIC ANALYSIS FOR SECURE INFORMATION FLOW AND VERIFICATION OF SOFTWARE SYSTEMS

Alan B. Shaffer
Commander, United States Navy
B.S., United States Naval Academy, 1986
M.S., Naval Postgraduate School, 1995

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2008

Author: _____
Alan B. Shaffer

Approved by:

_____
Mikhail Auguston
Associate Professor of
Computer Science,
Dissertation Supervisor

_____
Cynthia E. Irvine
Professor of Computer Science

_____
Gurminder Singh
Professor of Computer Science

_____
Gordon H. Bradley
Professor of Operations Research

_____
Timothy E. Levin
Research Associate Professor of
Computer Science

Approved by: _____
Peter Denning, Chair, Department of Computer Science

Approved by: _____
Doug Moses, Associate Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Within a multilevel secure (MLS) system, flaws in design and implementation can result in overt and covert channels, both of which may be exploited by malicious software to cause unauthorized information flows. To address this problem, the use of control dependency tracing has been explored to present a precise, formal definition for information flow. This work describes a security Domain Model (DM), designed in the Alloy formal specification language, for conducting static analysis of programs to identify illicit information flows, such as control dependency flaws and covert channel vulnerabilities. The model includes a formal definition for trusted subjects, which are granted extraordinary privileges to perform system operations that require relaxation of the mandatory access control (MAC) policy mechanisms imposed on normal subjects, but are trusted to behave benignly and not to degrade system security. The DM defines the concepts of program state, information flow and security policy rules, and specifies the behavior of a target program. The DM is compiled from a representation of the target program, written in a specialized Implementation Modeling Language (IML), and a specification of the security policy written in the Alloy language. The Alloy Analyzer tool is used to perform static analysis of the DM to detect potential security policy violations in the target program. This approach demonstrates that it is possible to establish a framework for formally representing a program implementation and for formalizing the security rules defined by a security policy, enabling the verification of that program representation for adherence to the security policy.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Above all, I wish to thank Dr. Mikhail Auguston, my dissertation supervisor. His endless encouragement and patient mentoring were vital to my completing this work. I thank the other members of my committee for guiding my research, in particular Dr. Cynthia Irvine and Tim Levin, who strengthened my work with their knowledge and expertise in the field of computer security. I would also like to thank Dr. Dennis Volpano for his suggested improvements to this paper, and Dr. Don Brutzman for his insightful advice at certain key moments when I needed a bit of "course correction."

For the past few years, I have had the good fortune of working alongside a wonderful group of fellow doctoral students. Our monthly "breakfast club" meetings were a source of true camaraderie, whether we were swapping war stories, telling bad jokes, or just helping each other survive the world of academic research. Their friendship and advice are truly appreciated.

I could not have reached this milestone without the love and support of my family. While Ashley prepared for her own transition from high school to college and Sam literally grew from young boy into young man, they dealt with a dad who (still!) spent his evenings and Sunday afternoons doing schoolwork. Throughout all of this, they never complained nor let me forget what is most important in my life. And to Sandra, who supported me without exception, I can never thank enough. At the most challenging moment, she offered the advice I needed most when she simply encouraged me to "get this thing done!"

Finally, I dedicate this dissertation to the men and women of the U.S. Navy's ships, squadrons and stations. They do the hard work of the fleet, and I hope this and future research can serve to make their jobs safer and more secure.

This material is based upon work supported by the Office of Naval Research, the National Science Foundation under Grant Number CNS-0430566, with support from DARPA ATO. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation of DARPA ATO.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

> *Working calmly will let you elaborate and extend things, but the breakthroughs generally come only after great frustration and emotional involvement.*
>
> — Richard Hamming

A. HYPOTHESIS

In the development of secure systems, ensuring that the implementation of a system is faithful to its stated security objectives is a process laden with difficulties and challenges. It is possible, however, to establish a framework for formally representing a program implementation and for formalizing the security rules defined by a security policy. This enables the verification of that program representation for adherence to the security objectives. Such a framework can be supportive of the Common Criteria requirements for secure system development.

B. INTRODUCTION AND MOTIVATION

Widely accepted evaluation standards (Common Criteria, 2006; DoD TCSEC, 1985), and (NSA SKPP, 2007) require that high assurance secure systems be designed, developed, verified, and tested using rigorous development processes. This evaluation process must include demonstration of correct correspondence between system representations at various levels of abstraction, such as security policy objectives, security specifications, and program implementation. In addition, the National Institute of Standards and Technology Source Code Security Analysis Tool Functional Specification (NIST, 2007) requires that security analysis tools report weaknesses that they identify using semantically meaningful names and identifying location within a program, with an "acceptably low false positive rate." This dissertation describes an approach to analyzing programs for preservation of security properties through state transitions, and advances the ability to analyze software for information flow by describing automated techniques for information flow static analysis. Classic work on secure information flow, including the use of lattice theory for ordering of security

classes based on the dominance relationship and the idea of labeling state variables to track such flows as a way to certify a program (Denning, 1976; Denning and Denning, 1977), and type systems for security analysis (Volpano *et al.*, 1996), provide a foundation for this research.

Formal security models are often based on an expression of properties such as program secure state and state transitions (McLean, 1994). High assurance evaluation standards (Common Criteria, 2006; DoD TCSEC, 1985) require a formal verification that the state transitions resulting from program execution preserve the security properties defined by a policy. Formal verification must also include advanced vulnerability analysis of a system, during which covert channel analysis must be considered in order to achieve successful evaluation of such systems at the highest levels of assurance (Common Criteria, 2006). The approach described here analyzes programs for preservation of specific security properties across state transitions. This dissertation presents a precise, formal definition for an MLS security policy, and for various types of information flow violations with respect to that policy, including examples of control dependency flaws and covert channels, extending classic work in this area by Denning and Denning (1977), and Volpano *et al.* (1996). A security domain-specific model is described as a framework for conducting static analysis of abstract representations of target program implementations (see Chapter VI).

Within a multilevel secure (MLS) system, *trusted subjects* may be granted privileges to perform operations, in some cases within prescribed limits (Schell *et al.*, 1985; Schellhorn *et al.*, 2000) not normally allowed for ordinary subjects controlled by mandatory access control (MAC) policy enforcement mechanisms. Granting of such privileges is predicated on the idea that trusted subjects will not conduct malicious activity or degrade the system's overall security. This dissertation presents a formal definition for trusted subject behaviors, which depend upon a representation of information flow during execution of a high-level language program, referred to as a *target program*. It describes a security domain model to formally represent security policy with respect to trusted subjects, trusted subject behaviors, information flow tracing through program execution, various types of covert channels, and a means for conducting

static analysis of target program implementations. In addition to trusted subjects and security kernels, we also include the analysis of programs (see Section VI.2) comprising interleaved statements of subjects with different labels. While this is not necessarily a literal representation of an application program, it can be viewed as the sequential actions that a trusted subject or security kernel takes in response to the interleaved requests of different subjects.

Static analysis of non-trivial programs has been shown to be undecidable (Rice, 1953). Even heuristics for static analysis may be computationally challenging, for example the problem of state explosion with model checkers (Clarke *et al.*, 1986). Jackson (2006) suggests a pragmatic approach to this dilemma, in the form of the Alloy language *small scope hypothesis*, which conjectures that most flaws in models can be revealed on small instances. In the approach described here, the Alloy Analyzer tool (Alloy, 2008) is used to perform static analysis of an abstract representation of a target program, referred to as a *base program*, to identify execution paths that might violate security policy rules. Our work assumes the small scope hypothesis for information flow tracing, and in the examples examined (see Section VI.C), a scope within the processing power of current technology was sufficient. The impact of this decision is that our approach is somewhat limited to evaluation of classes of programs for which proofs are not required. We feel that, as a proof of concept of the ability of our approach to perform automated static analysis, this is acceptable, and the work provides an important first step toward future advances that may overcome this.

This dissertation research is being conducted under the auspices of the Naval Postgraduate School's (NPS) Center for Information Systems Security Studies and Research (CISR) *Trusted Computing Exemplar* (TCX) project. The overarching goal of the TCX project is to present a working example to demonstrate how trusted computing systems and components could be constructed (Irvine *et al.*, 2004; Nguyen *et al.*, 2005). The goals of the TCX project are directly supported by the Office of Naval Research, the National Reconnaissance Office, and the National Science Foundation under grant CNS-

0430566. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ONR or the NSF.

C. CONTRIBUTIONS

This dissertation advances classic work in information flow tracing and static analysis verification of high assurance systems by presenting a new framework for automating the verification program representations for their adherence to a set of security policy rules. The framework is based on a domain-specific model (DSM) for security properties that supports static analysis of a representation of the target implementation. Our research delivers the following contributions:

1. The *Implementation Modeling Language* (IML), a language that supports basic information processing via assignment statements, conditional and loop statements, read/write statements, file random access, and access to a system clock. The IML facilitates static analysis of source code by providing a formalism that captures the essence of imperative programming language paradigms, while ignoring non-essential (for these purposes) elements, such as data type, inheritance, polymorphism, etc. In this research, a *target program* refers to the high-level language program under examination; the *base program* represents an IML abstraction of the target program and provides a basis for analysis for adherence to a security policy. At this time, the verification of correspondence of the base program to the target program is outside the scope of this dissertation.

2. The security *Domain Model* (DM), represented as an Alloy (Jackson, 2006; Alloy, 2008) specification, has a two-fold purpose in providing a model of program behavior, as well as a model for describing security properties. The DM is a unified representation of a base program and the intended information flow policy, including restrictions on both overt and covert information flow. The DM comprises an *Invariant Model*, which defines the logical structure for program state, information flow, and security policy; and an *Implementation Model*, which specifies the semantics of the base program.

3. This research has formalized well-established security properties by defining specific *security rules* in the DM framework. We have defined an *information flow* security rule to verify that information flows within a program abide by a policy and do not flow in an illicit manner. For example, a policy might authorize information to flow from a Low source to a High destination, but not the reverse; the information flow security rule will identify any program execution path that allows information to flow illicitly from Low to High.

In our framework, we have also defined rules for detecting potential *covert channels* in a program. In the context of the DM, we do this by formalizing the definition of both a covert storage channel, using a shared information storage repository, and a covert timing channel, based on access to an abstraction of the system clock. For each of these covert channel classes, we define a security rule that identifies programs with execution paths that may be vulnerable to either violation.

In addition to covert channels, we have formalized the concept of *overt control dependency flaws* within a program. Using *dynamic slicing* techniques to track control dependencies through execution of a program representation, the security rule identifies execution paths with implicit control dependencies that violate the defined security policy.

Finally, our model is expressive enough to support *trusted subjects* and their behaviors. The DM defines a trusted subject through a special assignment operation that abstracts the idea of trusted downgrading. The model allows trusted subject behaviors, while ensuring that regular subjects cannot perform illicit activities during execution of a program. We do this through verification of the described security rules.

4. We have implemented a prototype for static analysis of programs based on the DM framework. This framework includes a specialized *DM-Compiler*, developed to translate a base program written in IML into an *Implementation Model* and to integrate it with the *Invariant Model* to form a complete DM specification to represent the original *target program*. This prototype implementation architecture is depicted in Figure 1, and will be explained in detail in Chapter V.

5. Early results in this dissertation work were presented at several conferences and workshops. Our paper presented at the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07) workshop on Domain-Specific Modeling (Shaffer *et al.*, 2007) demonstrated the ability of the DM approach to detect illicit information flow violations. At the ACM Conference on Programming Language Design and Implementation (PLDI) workshop on Programming Languages and Analysis for Security (PLAS'08), we presented our work on analysis of programs for detection of covert channels, and overt flaws based on control dependency analysis (Shaffer *et al.*, 2008). We presented our implementation of trusted subjects and their behaviors at the Modeling Security Workshop in association with the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'08) (Shaffer *et al.*, 2008). At the International Conference on Software Engineering and Knowledge Engineering (SEKE'08) (Shaffer *et al.*, 2008) we presented a survey of our overall approach to static analysis using the DM, and have submitted a comprehensive article discussing the final stage of our research for inclusion in the *Computers and Security Journal* special issue on "Software Engineering and Secure Systems" (Shaffer *et al.*, 2008).

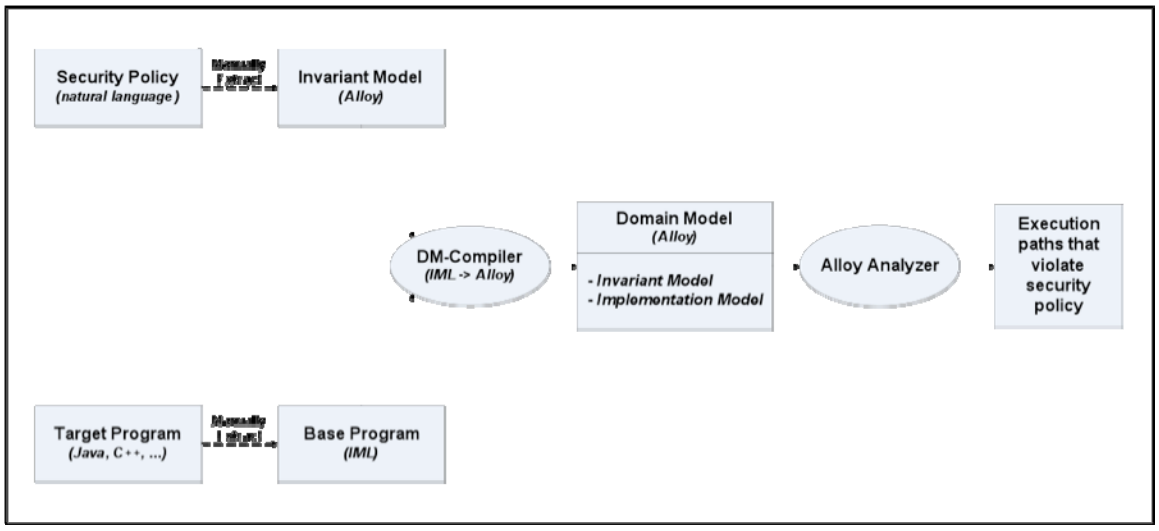

Figure 1. Domain Model approach to system security verification.

In addition to these publications, we have built a Security Domain Model Project website (http://cisr.nps.edu/projects/sdm.html) that provides discussion of the research, with links to our papers, and example IML base programs with respective DM specification code.

6. Relevance to Department of Defense (DoD). Highly secure systems are significantly more expensive to build than typical IT systems. because a contributing factor to these high costs is the need for formal verification, which requires a large amount of effort by highly trained specialists. The contributions described in this work advance the ability to automate a portion of the formal verification process. This automation has two significant effects: it has the potential to reduce errors caused by human mistakes in the verification process, making systems more secure; and it can reduce the effort required for verification, making secure systems more affordable.

These techniques are applicable to the verification of a specific class of components (target programs) in secure MLS systems, for which the DoD has projected both a significant need and future support of extensive research and development (NSA GIG, 2005). These components perform the following abstract function:

- Given a machine with two input ports (labeled HIGH and LOW) and two output ports (also labeled HIGH and LOW), the machine processes inputs and sends output only to the *appropriate* output port, where "appropriate" is defined by the security policy.

The framework developed here supports automated static analysis to determine whether the system component in question conforms to, or enforces, its security policy. The framework does this by: (1) associating with each incoming datum an internal label equal to the label of the input port, (2) modifying the internal label of a datum, if necessary, as it is processed, and (3) detecting if a datum leaves the component through an output port that is *inappropriate* with respect to the datum's internal label and the output port label.

Examples of such components are trusted subject programs, security kernels, and separation kernels configured to enforce an MLS policy. Trusted subjects are often used

in MLS systems as downgraders, multilevel switches, and for other critical functions that make MLS systems useful and effective. Security kernels and separation kernels are the foundation of most DoD systems that provide multilevel security. The DoD, working with other government agencies, is researching systems for sharing information across differing security levels, under the aegis of the Unified Cross Domain Management Office (UCDMO) (Thuermer, 2007).

D. DISSERTATION ORGANIZATION

An outline for the remainder of this dissertation follows:

- Chapter II presents an overview of information assurance principles germane to this dissertation research. Specific discussions include high assurance system evaluation processes and standards, formal security models and policies, multilevel secure (MLS) systems, information flow tracing, covert channels, trusted subjects, and an introduction to dynamic security policies.

- Chapter III provides an analytical review of previous work related to the fields of the dissertation, to include background research on security models and policies, analysis of systems for illicit control dependencies and covert channels, implementation of a trusted subject into secure systems, and development and implementation of systems which implement dynamic security policies.

- Chapter IV presents the Implementation Modeling Language (IML), developed as a specialized language for representing target program implementations as base programs.

- Chapter V presents the Security Domain Model (DM) framework and an associated approach to performing static analysis of base programs to verify their adherence to security policy rules.

- Chapter VI presents several example base program test cases, each with security vulnerabilities discoverable using the DM approach, and provides an analysis of results of the testing.

- Chapter VII presents conclusions of this research, and suggests areas of future work.

- Appendices are included at the end of this dissertation to provide RIGAL compiler construction language code for the *DM-Compiler* files, and several example base programs with complete Alloy specifications for each associated Security DM.

THIS PAGE INTENTIONALLY LEFT BLANK

II. INFORMATION ASSURANCE PRINCIPLES

A. INTRODUCTION

This chapter discusses concepts and principles, within the information assurance domain, germane to this research. It begins with an introduction to high assurance system evaluation, and the established standards that govern this process. A discussion of formal security models and policies follows this. Next, we introduce the concepts of information flow through program execution, and control dependencies which can affect such flows. The concepts of covert channel analysis, and trusted subjects and their behaviors are then discussed. Finally, dynamic security policies are introduced, as an area for future work in this research.

B. HIGH ASSURANCE COMPUTER SYSTEM EVALUATION

1. Introduction

The Common Criteria (CC) is a recognized International Organization for Standardization (ISO) standard for evaluating computer systems against varying levels of assurance. Initially published in January 1996, the CC essentially combined the older and independent U.S., European and Canadian standards for system evaluation, incorporating the best principles and tenets of each. While the CC was also intended to replace the older Department of Defense (DoD) Trusted Computer Security Evaluation Criteria System (TCSEC), the de facto U.S. standard since the mid 1980s, a few proponents of the TCSEC continue to promote it as a superior standard. Thus, while the CC is the official evaluation criteria used by the U.S. government and DoD, both standards have their advocates within the security community.

2. Trusted Computer Security Evaluation Criteria System (TCSEC)

Developed by the DoD as part of the *Rainbow Series* of computer security publications, the TCSEC (commonly referred to as “the Orange Book” because of the color of its cover page) was initially drafted in 1983, and accepted in 1985 as the US standard for computer systems assurance evaluation and classification (DoD TCSEC,

1985). The TCSEC defined a range of hierarchical assurance levels, based upon the effectiveness of security provided by and within a particular system being evaluated, such that a given level includes all of the requirements of a lower level, and more. The assurance levels increase from level D through level A, with the higher levels (C, B, A) further divided into classes based on the specific protection characteristics provided. These levels represent the assurance classification for a system, from one evaluated to have minimal or no security capabilities (level D), up through one that has undergone formal methods verification that its protection system and model are correct, and whose implementation corresponds to the formal top-level specification (level A1).

The TCSEC required the identification of security properties such as those identified by Bell and LaPadula (1973) in their security model of mandatory access control, and was based heavily on the reference monitor concept for adjudicating accesses of subjects to objects within an operating system (Anderson, 1972). Also, the Orange Book focuses on operating system security, while the need clearly exists to account for security of many other types of systems (in fact, other books in the *Rainbow Series* were written specifically to address non-OS computer systems). These shortfalls, coupled with the lack of an international standard, provided the impetus for development of an encompassing standard for system security evaluation.

3. The Common Criteria

Schell (2001) defined four epochs spanning the history of scientific developments in computer security, each one progressing beyond the previous toward better security and more assured systems – that is, each epoch but the last. This final (and current) epoch Schell defined as being one of a period of security decline, with the Common Criteria as its hallmark. He pointed to the fact that, while the CC evaluates individual products, the science of computer security should instead focus on evaluating entire systems, which comprise multiple software and hardware products. Schell believed that by making no distinction between system and subsystem evaluation, the CC limits us to examining only isolated components of any system, never evaluating the system and its IT environment in their entirety.

Schell's negative assessment notwithstanding, the CC represented a paradigm shift in the security evaluation process, and has become the US and international standard for the development of computer system security specifications. The CC allows evaluation of a specific system (or product), or more generally, families of systems (or products), by establishing security assurance criteria against which they can be measured. The CC introduced the concept of a Protection Profile (PP) to describe the security requirements of a category of products. A PP is typically written to define the requirements for a class of system by some user community, and it generally defines the requirements for both functionality and assurance of the system. A Security Target (ST), on the other hand, is used to evaluate a specific system or product. An ST is often written based on an existing PP for a like class or category of system, in order to define the requirements of the *target of evaluation* (TOE) (Common Criteria, 2006).

The CC defines a number of assurance *classes* within which security criteria and requirements are defined. Among the assurance classes are such topical areas as development, life-cycle support, and testing, for example. These broad classes are subdivided into *families* that further describe the unique objectives of the class. As an example, the Development class (identified as "ADV") contains a family called Security Architecture (uniquely identified as "ADV_ARC") whose objective is to allow "the developer to provide a description of the security architecture of the TSF" (Common Criteria, 2006). Finally, each family is defined by some number of hierarchically ordered *components*, each of which describe the scope, depth and rigor required of a security criterion, in order for the product to meet a particular assurance level.

The goal of the CC in defining assurance classes is to allow evaluators to measure a system against the objectives described by the families of these classes. For this, the CC defines Evaluation Assurance Levels (EALs) which "provide an increasing scale that balances the level of assurance obtained with the cost and feasibility of acquiring that degree of assurance" (Common Criteria, 2006). The CC EALs were written with the intent that they correlate roughly to the TCSEC Orange Book grading levels, primarily so

that earlier evaluated systems could remain relevant. Since the two systems evaluate assurance in different ways, however, their respective grading levels should not be regarded as equivalent.

For a system to achieve a given EAL, it must meet the security criteria for all of the family components of that level. To be evaluated at the highest CC assurance level – EAL7 – a computer system must undergo a more comprehensive analysis than that required for lower assurance levels. This evaluation analysis must include formal representation of system requirements, demonstration of formal correspondences, and comprehensive testing of system components. In particular, the formal correspondences must be demonstrated as mappings between the security policy model and the functional specification of a system.

Further, a complete correspondence must be demonstrated between the system security objectives and functional requirements, and between the functional specification and functional requirements. In general, with respect to the implementation of the system, a full mapping must be shown to exist between all levels of design description and specification, as depicted in Figure 2. This formal correspondence must be proven between all constructs of the system in order for a high-assurance system to be evaluated at EAL7.

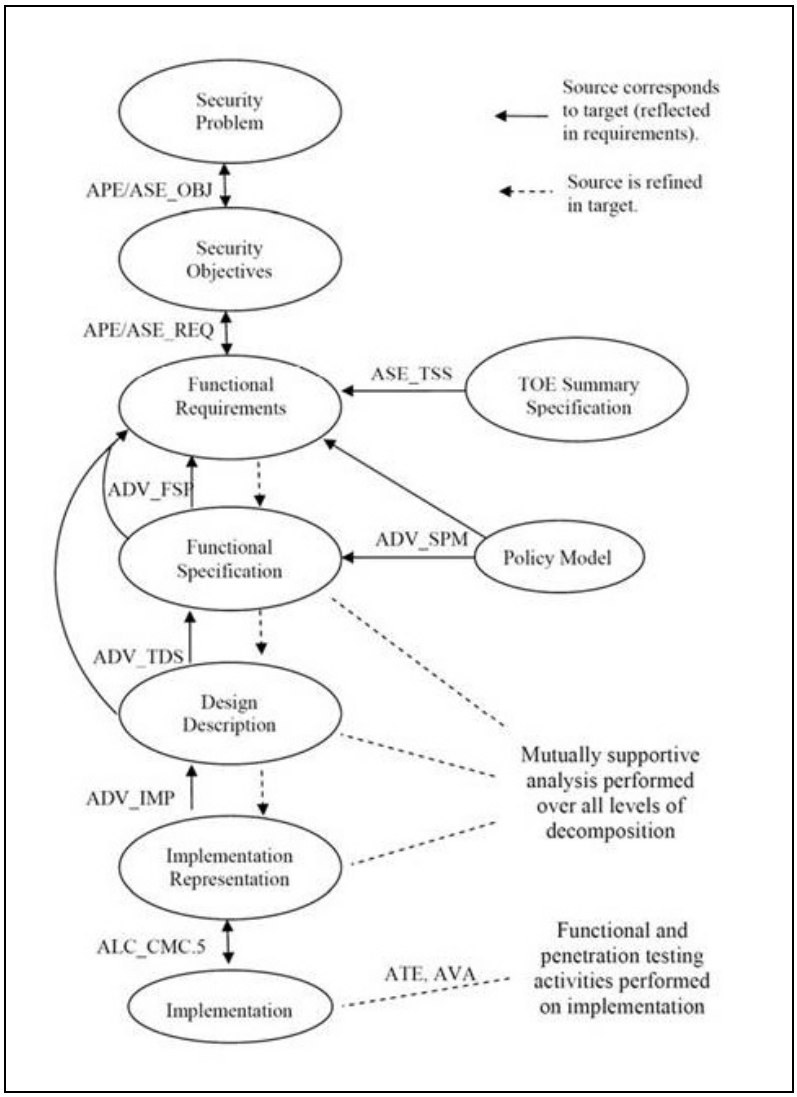

Figure 2. Common Criteria development class (ADV) correspondences (from Common Criteria, 2006).

4. The Trusted Computing Exemplar (TCX) Project

The Trusted Computing Exemplar (TCX) project is being conducted by the Center for Information Systems Security Studies and Research's (CISR) at the U.S. Naval Postgraduate School (NPS) in Monterey, California. The goal of TCX is to demonstrate in an openly distributed, worked example, how a trusted computing system and its components can be built to meet high assurance evaluation criteria. Its specific approach to meeting this goal is fourfold (Irvine *et al.*, 2004):

- Create a prototype framework for high-assurance system development
- Develop a trusted computing component as a reference
- Evaluate the component, through a third-party, against CC EAL7
- Provide a means for open distribution of all deliverables and artifacts

To date, the TCX project has achieved milestones in defining a high-assurance development framework, and the design specification for a web-based dissemination system (Levin *et al.*, 2004; Nguyen *et al.*, 2005). Its least privilege model has been articulated, and features to support transitive trust have been described. The project has created a working prototype and is developing design and implementation of the TCX Least Privilege Separation Kernel (TCX-LPSK) as a trusted component. By developing a framework of application mechanisms for specifying and verifying the mapping of higher level security requirements for a system, this dissertation research directly supports the goals of the TCX project.

C. FORMAL SECURITY MODELS AND POLICIES

1. Introduction

McLean (1994) stated that security models are "used to describe any formal statement of a system's confidentiality, availability, or integrity requirements." Security models provide a detailed and precise means of formally describing security policies, and proving their validity. Since a system must not only be secure, but demonstrably so, formal security models provide system designers with evidence that they are constructing a self-consistent system, and with a foundation for further demonstration that the system as implemented meets its specifications (Landwehr, 1981).

In a typical formal methods approach to security model development, a security policy is initially translated from words into a mathematical model. From this, a formal specification is created and shown to satisfy the mathematical model, deriving from the original policy. To prove conformance to the policy, the specification must be implemented and shown to be an accurate representation of the specification. This "chain of correspondence" may not be sufficient, however, to ensure a secure system as various

aspects of security may be outside the scope of the formal security policy model (for example, physical security and good password hygiene).

While it would be difficult to provide an exhaustive review of every security model that has ever been developed or proposed, we attempt to provide here a representative overview of some well-known ones. This analysis will divide security models into two logical categories: discretionary access control (DAC) models and mandatory access control (MAC) models. Generally, DAC systems allow one user to extend to other users or subjects his rights to the objects for which he controls access. In DAC systems, the ability to modify access rights is subject to some set of rules which can change during the course of system operation. Conversely, MAC systems attempt to provide a global and persistent policy, where access control is fixed in a predetermined state by a predefined set of rules and all modifications are relative to that established set of allowed access rights.

2. Discretionary Access Control (DAC) Models

a. Access Matrix Models / Graham-Denning

In the context of computer operating systems, Lampson (1971) defined protection as "all the mechanisms which control the access of a program to other things in the system." To describe how this control of access would occur, he developed the concept of using matrices as a way to illustrate the access of subjects to objects within a computer system.

An access matrix scheme is comprised of *objects* and *subjects* formed into a matrix of allowed accesses. The objects can represent anything in the system that needs to be protected, including files and processes. Subjects represent system entities that can have access to objects. The access matrix scheme also includes a set of rights representing the types of accesses which a subject may have to an object, for example read, write, execute, or owner.

In general, an access matrix defines a cross-mapping between the objects and subjects in a system. Cells in the matrix represent monitors that control access of a particular subject to a particular object. A given access matrix reflects a security policy

for the system. A "snapshot" of the current access matrix can be thought of as a specific protection state of the system, and defines the access rights in that state; to modify or add to the set of allowed accesses would require some transition into a new matrix, or protection policy.

The *Graham-Denning Model* (1972) formalized the concept of using access matrices for access control within operating systems. Their model defined basic access rights of a system in terms of a set of eight allowable operations across subjects and objects within that system. These operations permitted a subject *x* to create and delete objects and subjects, and to read, grant, delete and transfer the access rights of another subject *s* to an object *o*. The Graham-Denning model is generally considered to be the first formal DAC scheme proposed and the first general access matrix model.

An access matrix is a static representation of a security system, and corresponds to a set of access rights for a given security policy. Rules might be established to define how subject-to-object access rights could be modified over time, possibly allowing users the discretion to extend rights to other subjects. Changes to the system from such rules would transition the system into a new state, essentially representing a new security policy.

b. HRU Model

The *Harrison-Ruzzo-Ullman (HRU) Model* (Harrison *et al.,* 1976) defined authorization systems that allowed the modification of access rights, along with the ability for creating and deleting subjects and objects within the system. It also introduced a *safety property*: that access to an object within the system was impossible without the concurrence of the owner of that object.

In reality, since an owner in a DAC system may extend rights to an object that in turn may be given away without his knowledge, no protection system can be safe by this definition. To solve this problem, the HRU model provided a weaker definition that simply required a protection system to ensure that objects are kept "under control" by their owner, meaning that the owner has some intuition as to whether his granting of a right could lead to possible leakage of that right to an unauthorized subject in the system.

Even this weaker definition of safety is too strict for all protection systems, since it is generally undecidable whether, "given an initial access matrix, there is some sequence of commands in which a particular generic right is entered at some place in the matrix where it did not exist before." They proved that, while an algorithm could be found to show whether a given mono-operational system is unsafe for some generic right, it is not possible to devise an algorithm to decide the safety of any generic protection system for all of its possible configurations.

3. Mandatory Access Control (MAC) Models

Mandatory access control models are representative of multilevel systems in which subjects and objects are hierarchically or partially ordered according to their sensitivity levels, and the system ensures that data from objects higher, or non-comparable, in the hierarchy is not available to subjects lower in the hierarchy. One of the most common examples of a MAC structure is the military classification system.

a. Bell and LaPadula Model

The *Bell and LaPadula (BLP) Model* (Bell and LaPadula, 1973) can be used to formally describe the enforcement of military policies associated with classified information. At the heart of the BLP model is the concept that all objects in the system are assigned a classification level based on their relative sensitivity, and all users are similarly assigned a clearance level based on their job, rank, experience, etc. System subjects (users) are granted access to objects (files, applications, resources, etc.) based on the relationship between the clearance level they possess and the defined classification level of the object.

The BLP Model defines the *simple security (ss-) property*, which states that a subject at a given security level can only access objects assigned an equal or lower classification level, commonly referred to as "read down". Similarly, they define the *confinement (*-) property,* which prevents sensitive information from being copied into less sensitive objects by malicious software, by ensuring that a subject cannot write into an object that is of a lower classification level; this rule prevents "write down". These

rules allow (and restrict) the flow of information across the multiple levels of the security structure, thus the BLP model and its derivatives are popular information flow models.

The BLP model can be described by a set of rules governing the relationship between objects and subjects, and the associated access functions over them. Stated more formally these rules, or properties, provide the most common way of representing the model:

- The *simple security (ss-) property* states that every "observe" access triple (*subject, object, read right*) in the current access set *b* must have the property that the level of the subject dominates the level of that object.
- The **-property* states that, if a subject has simultaneous "observe" access to object O_1 and "alter" access to object O_2, then the level of O_1 is dominated by the level of O_2.
- The *discretionary security (ds-) property* states that, if (subject *i*, object *j*, attribute *x*) is in the current access set *b*, then *x* is recorded in *ij*'th component of the access matrix *M*.

As an example of the BLP model, suppose a system user has been assigned a clearance level of "secret." This means that the user possesses the access rights to read documents that are classified at the secret level and lower while logged onto the system at the secret level, and to write to secret and higher documents (although the user would not have read access to higher classified documents in this system, in order to write to them in a coherent manner).

The rigidly defined structure of the BLP model is both its strength for protecting military classified information, and its most commonly highlighted weakness. In particular, enforcement of the **-property* does not allow an object to be downgraded to a lower classification level. While this is a basic requirement of a practical MLS system, the **-property* prevents it. To address this shortcoming, Bell and LaPadula introduced the idea of a *trusted subject*, which is one that can be trusted not to violate the intent of the **-property*. Critics have also pointed out that the BLP model could allow a malicious user to simply request the system administrator to temporarily declassify a file, thus allowing a low user to potentially read a high file. To address this criticism, Bell and

LaPadula added the so-called *tranquility property* (Bell and LaPadula, 1976). The *strong tranquility property* states that the security labels of objects can never change during system operation, while the less stringent *weak tranquility property* stated that the labels can never be changed in such a way that violates any of the other defined security properties.

b. Biba Integrity Model

The BLP model was concerned only with confidentiality of objects. While this was intended by design, a follow-on information flow model was developed that specifically classified data according to integrity levels; here, integrity can be thought of as the quality of the information, for example, stored in an object. The *Biba Integrity Model* (Biba, 1977) can be thought of as complementary to the BLP model: whereas BLP focuses on the sensitivity of objects, Biba defines its rules based on maintaining their integrity. With the proliferation of digital data storage, there is good reason for a policy that ensures that the near- and long-term integrity of data is maintained.

The goal of the Biba model is to ensure that high integrity documents will not be contaminated by lower integrity data. In doing this, Biba ensures that a subject is only allowed to view objects of an equal or higher integrity level than itself. Under the Biba model, individual integrity levels are defined for all subjects and objects in the system. The higher the integrity level, the higher the level of confidence, reliability and trustworthiness in that entity (this concept has nothing to do with an object's inherent sensitivity). The rules of the Biba model define a strict integrity policy which prevents information flows from lower to higher integrity levels, and can be formalized by the following axioms (note that the differences between these and the properties that define the BLP model are symmetric):

- The *simple integrity (si-) axiom* states that a subject *s* can read from an object *o* iff the integrity of the subject is dominated by that of the object; in other words, a subject cannot read from an object of lesser integrity ("no read down").

- The *integrity *-property axiom* states that a subject *s* can have write access to an object *o* iff the integrity of the subject dominates that of the object; in other words, a subject cannot write data to a higher integrity level than its own ("no write up").

While the concepts underlying the Biba model seem straightforward, particularly when understood as the dual of BLP, Biba has been difficult to implement in real-world systems. However, integrity policies have been investigated in some high assurance systems, such as GEMSOS (Schell *et al.*, 1985), which implemented Biba controls using integrity labels. The general difficulty in implementing Biba lies in how one defines *integrity*. Whereas the sensitivity level for subjects and objects can be readily defined and assigned – one only need consider the military classification system as a practical example of this – the same cannot be said for assigning integrity levels.

Establishing criteria for assignment of various integrity levels to subjects and objects has proven difficult. What parameters should be taken into account when assigning an integrity level to a user? Integrity for one class of users may be defined very differently from that of another, for example the integrity of medical records is quite different from that of the source code for a popular video game. In addition, Irvine and Levin (2001; 2002) pointed out the "integrity problem" of a system based on its *integrity capacity*. They showed that, in order to ensure the integrity of all its objects, a system is practically limited to maintaining (modifying) only data whose integrity is as low as its lowest integrity component. For implementation of integrity policies in high-assurance systems, Schellhorn *et al.* (2000) have suggested using Common Criteria EAL levels, for example, as integrity labels when storing applications to smart cards, however, this requires that evaluation criteria for these applications must always be compatible.

c. Lattice Model

The *Lattice Model* is another in the class of mandatory access control models, and is based on the idea of classes of objects being organized into a universally bounded lattice. Developed by D. Denning (1976) as "a structure consisting of a finite partially ordered set together with least upper and greatest lower bound operators on the set," her goal was to find a model with suitable restrictions such that its security would be

not only decidable, but *simply* decidable. This would ensure a system with proven ability not to leak information from an object at a high sensitive level to a subject at a lower sensitivity level.

For her lattice model, Denning defines the flow operator "→" acting on a pair of object classes *X* and *Y*, such that *X*→*Y* indicates that information is permitted to flow from an object in *X* to one in *Y*. In this sense, "information flow" means that information associated with one class affects the value of information associated with the other. Figure 3 shows an example of a three-dimensional universally bounded lattice. The lattice represents a set of properties {*x, y, z*} in a system, and establishes a hierarchical dominance relationship among the set and its subsets. Information in one object can only flow into another object if the second possesses at least the properties of the first; that is if the second dominates the first. For example, an item of information contained in class *x* could flow into {*x, y*} or into {*x, z*}, but could not flow into class {*y, z*}. Similarly, information contained in class {*y, z*} could not flow into class {*x, z*}, since *y* is not contained in the class {*x, z*}.

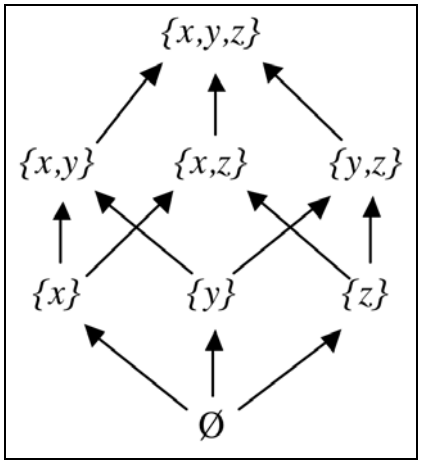

Figure 3. Lattice of Subsets for {x, y, z} (from Denning, 1976).

Objects within the lattice can be defined by any ordering, and the lattice may be defined with categories non-comparable to one another in ordering objects. This does not preclude, however, a lattice from being ordered by a combination of sensitivity

or integrity, provided information flows were properly defined to ensure that the *ss/si-properties* and the **-property* were adhered to (Lunt *et al.*, 1990).

d. Clark/Wilson Model

As with the previous examples of MAC models, the *Clark/Wilson Model* (Clark and Wilson, 1987) is an information flow security model which, like the Biba Model, ensures the integrity of objects. Clark and Wilson stated that prevention of unauthorized disclosure of sensitive information, which is vitally important in military security, is less important in commercial applications where integrity of information and prevention of unauthorized data modification is paramount. Loss or corruption of a company's records and stored data through fraud or errors is often the gravest danger.

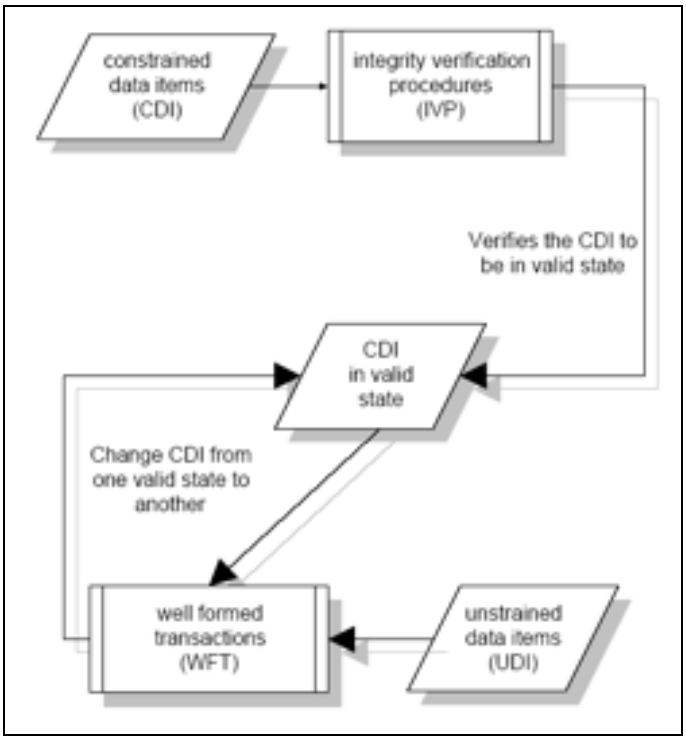

Figure 4. Clark/Wilson Model of Integrity (from Neumann, 1998).

The C/W model addresses the flow of information by relying on software application *well-formed transactions* (WFT), and separation of duties among users to ensure that no single user can contaminate a data object through unauthorized changes. WFTs ensure that no user can arbitrarily modify data, either maliciously or unintentionally, and can only act in constrained ways that ensure data integrity. In addition, strict logging of all user actions facilitates auditing the system for erroneous or malicious transactions.

In addition, the C/W model further defines *constrained data items* (CDI) as those objects requiring integrity; *integrity verification procedures* (IVP) as applications procedures which provide confirmation that all data adheres to security specifications on the system; and *unconstrained data items* (UDI) as those objects outside the system that have yet to be verified for integrity.

Figure 4 provides a simple example of data object flow through a C/W system. In this snapshot of the system, all CDIs have previously been verified for integrity by an IVP, thus placing them in a valid state. Any proposed manipulation on a CDI must be performed via a WFT, which will ensure the CDI's return to a valid state. Introduction of a new UDI into the system will require verification of its integrity through an IVP before it can be reclassified a CDI and placed into a valid state.

Shockley (1988) showed how the C/W model could be partially implemented as an MLS system, using BLP mechanisms, and Ge *et al.* (2004) showed the implementation of the C/W model using a DBMS server. Since the C/W model relies heavily on application procedures for its WFTs, these procedures must be proven valid for all possible states. Otherwise, there is no assurance that a WFT returns a CDI to a new valid state after the transaction.

e. Noninterference

Some approaches to secure information flow do not distinguish between classes of covert channels, or between covert and overt flows for that matter. These approaches rely on the concept of *noninterference*, which states that the actions of one subject can have no effect on the output of a lower subject in a system. Goguen and

Meseguer (1982) described a model wherein security policies are defined in terms of only noninterference assertions, rather than by the combination of access control and covert channel restrictions. In their research on security policies and models, they described noninterference within a system in terms of information flows permitted between high and low subjects. They explained that "one group of users, using a certain set of commands, is *noninterfering* with another group of users if what the first group does with those commands has no effect on what the second group can see." Haigh and Young (1987) further expanded the noninterference ideas of Goguen and Meseguer in follow-on work.

The noninterference concept, when initially proposed, appeared to be a logical description of confidentiality in the context of an MLS system, however, problems were soon realized in implementing noninterference. In practice, high-level data often has an effect on low-level data, such as the case where a highly classified file is encrypted or declassified, and then transmitted over a low channel. An activity such as this would represent a low user receiving information (declassified data) based on the actions of a high user (the downgrader), and would not be allowed under the strict definition of noninterference. In addition, noninterference had a significant shortcoming, with respect to BLP, in that it did not prevent "read up" of high sensitivity objects by a low sensitivity subject. While noninterference ensures that the actions of a high level subject will not affect low level outputs in a system, it does nothing to prevent a low level subject from somehow observing high level information, and then acting on that information.

The access control models described above (the BLP Model, etc.) are used in formal verification to prevent *Low* subjects from accessing *High* information, using a two-part strategy: (1) by enforcing rules regarding how subjects may access objects; and (2) by performing covert channel analysis to close remaining illicit flows. While noninterference was ostensibly proposed as a way to define information flows, in practice it was also seen as a better way to explain covert channels, since access control did not adequately do this, and in fact, noninterference sought to unify the concepts of access control and covert channel analysis.

Noninterference with respect to security properties is often considered limited by the *refinement paradox*, which states that a system's abstract security properties for information flow cannot be guaranteed to be preserved through refinement to concrete implementation (McLean, 1990; 1996; Roscoe, 1995). Also, in their report from the 2001 Computer Security Foundations Workshop, Ryan *et al.*, (2001) stated that noninterference, as a way to ascertain covert channels (due to algorithm or design flaws), does not pass the *necessary and sufficient* test. A covert channel does not necessarily imply the presence of interference on a practical level. That is, while interference is *sufficient* to imply a covert channel, it may be so small as to be of no practical exploitable use. The report stated, "in most noninterference models, a single bit of compromised information is flagged as a security violation, even if one bit is all that is lost. To be taken seriously, a noninterference violation should imply a more significant loss." The ability of noninterference to adequately contain information flows and covert channels, as well as its overall validity as a security model, has been hotly debated in the computer security field.

4. Role Based Access Control (RBAC) Model

RBAC security models represent a hybrid class of non-discretionary access control security policies that were designed to provide for the security requirements of non-military organizations. RBAC was developed at the National Institute of Standards and Technology (NIST) to meet the needs of industry and civilian government organizations, where the strict requirements of a military MAC policy do not address the security needs for handling sensitive unclassified information (Ferraiolo and Kuhn, 1992; Sandhu *et al.*, 1996). This type of environment can be exemplified by a medical facility, where personal patient information can often be extremely sensitive, although not classified in a strict sense. In this type of environment, specific information should be handled by only qualified classes of users, based on the sensitivity level of the information. As an example, while medical charts and patient history are vital information that doctors and nurses must access to effectively do their jobs, such data need not (and should not) be made available to hospital administrative staff, who have no such valid access requirement. The opposite would be true of financial information

concerning a patient's ability to pay for medical care services. In each case, the role of the user represents the key factor in determining whether or not information should be made available.

To address this type of security environment, an RBAC model grants access rights to groups of users based on their role requirements, rather than providing rights to users individually. Essentially, RBAC is a variation of an access matrix model, where groups of subjects represent roles, and rights are assigned uniformly to the group, as opposed to individual users (subjects). Each role defines a specific set of operations that the individual acting in that role may perform, or in terms of access rights, the set of objects a user has access to, based on his role.

The military classification system's *compartmentalization* labels for classified messages may be viewed as another application of RBAC. In addition to hierarchical classification labels representing secrecy level, messages may also be tagged with a compartment label, e.g., NUCLEAR, which can be viewed as analogous to a *role*. In this scheme, users are granted access to messages with a particular compartment label (assuming they also possess adequate clearance level), based on their job requirements. At a later time based on changing job requirements, a user may be debriefed out of a particular compartment, essentially losing this role, at which time that user would no longer have access to messages with this compartment label.

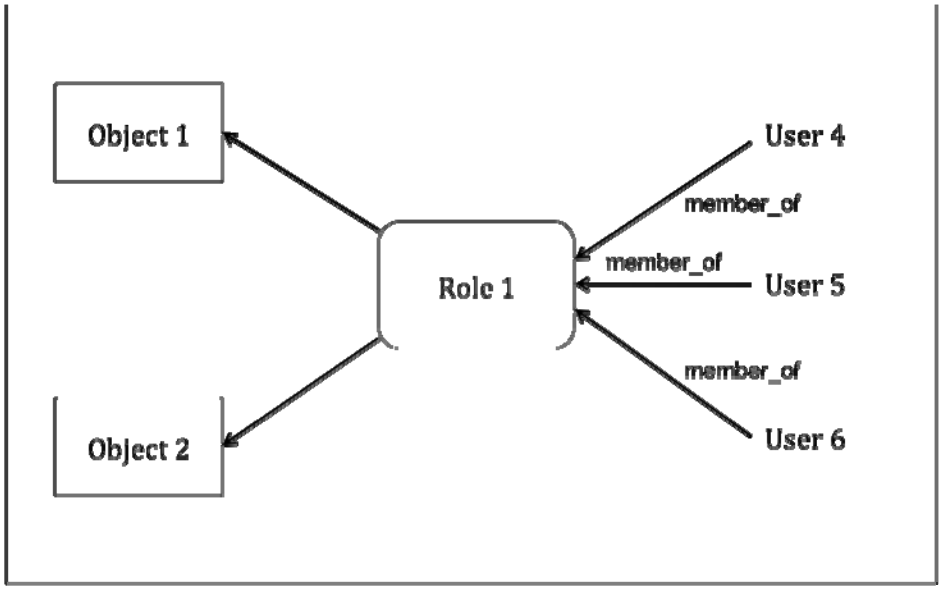

Figure 5. RBAC Role Relationships (after Ferraiolo and Kuhn, 1992).

Figure 5 shows a simple relationship between users, roles, and objects. In the example, *User4* is identified as possessing *Role1*, presumably by virtue of his job description. Because he possesses this role, he implicitly has been granted access to *Object1*. If at some time in the future *User4* were to lose possession of *Role1*, he would no longer have access to *Object1* (note that this change would require no modification whatsoever to *Object1*, nor to *User5's* or *User6's* access to it).

The RBAC model provides the means to readily support the principle of least privilege at the granularity of the role or group. This principle states that a subject will have access to only the objects needed to perform his or her job, and nothing more. In an RBAC model, specific rights and privileges can be granted to an entire group based on the required functionality of their role. The administrative strength of RBAC is that privileges can be granted (or retracted) to the group as a whole, without the need for manipulation of individual user rights.

5. Summary

This section has described well-known mandatory and discretionary access control security models, intended to protect both confidentiality and integrity of information. Because our research focuses on MLS systems, particularly systems

managing military classification information, we have designed our approach using a Bell and LaPadula type of security policy. As will be described later in this dissertation, the BLP simple security and *-properties are formalized as security rules against which programs are verified for security. However, our model can facilitate expanding the security policy to capture integrity policies, for example by defining rules associated with the Biba model, or by defining information "compartments" using RBAC-style roles to adjudicate access.

D. COVERT CHANNELS

As defined in early research by Lampson (1973) and Kemmerer (1983), covert channels use system properties not intended as communication channels as a way to transfer information between system subjects. Such channels allow processes to take advantage of communication channels to transfer information in a manner that violates a security policy.

An operating system may virtualize a shared physical resource so that each subject, or equivalence class of subjects, perceives that it has exclusive access to the resource. A covert channel can result from the incomplete virtualization of a resource such that some *attribute* of the resource remains shared, indirectly.

Schaefer *et al.*, (1977) defined covert channels as being either *storage* or *timing* channels. For both storage and timing channels, the sender and receiver (typically subjects) must have the following capabilities (Kemmerer, 1983):

1. Indirect access to an attribute of a shared resource, which the sender can modify, and the receiver can view. For example, the shared resource is the CPU, and the attribute is its "busy" state; or the shared resource is the disk, and the attribute is the location of the disk arm, or the attribute is the "full" state (Karger and Wray, 1991).
2. A means to initiate and synchronize their actions. The sender and receiver need to know when to modify and observe the attribute, the importance of which increases when they wish to transmit a stream of data.

This dissertation research essentially distinguishes between a covert storage channel and a covert timing channel by the means in which the receiver observes the change in the attribute:

1. Storage – the receiver views an error message, or other information placed in its address space by the system, for example if a storage disk is filled the receiver is provided an error message to that effect.

2. Timing – the receiver views changes to the relative timing of "legal" events. For example, if the sender's activity makes the CPU busy, the receiver's request to execute an operation on the CPU will complete (event 1) after the expected time of day occurs (event 2). Or, in the case of the disk arm attribute, depending on where the sender has left the arm (by reading a sector near the inner or outer edge of the disk), two disk sectors read by the receiver will occur in a different order (events 1 and 2).

Goguen and Meseguer (1982) defined a *point of interference* between two subjects as the point where the *high* subject interferes with the context of the low subject. An exploitable covert channel can result in information flow in violation of the intended security policy. The point of interference of a covert channel is considered an *internal resource* of the system, as it is not directly accessible to subjects, as are *exported resources* (NSA SKPP, 2007). Note that if a *low* subject can directly view the value of an exported resource (such as a variable) that has been modified by a *high* subject, then an overt flaw rather than a covert channel results.

In the case of a mandatory access control (MAC) policy, the covert channel sender's sensitivity level will be higher than the covert channel receiver's sensitivity level, with respect to confidentiality (Kemmerer, 1983). Thus, the determination of the potential covert channels in a system depends not only on the policy in place, but also on the implementation of that policy on a specific system (Gligor, 1993). Our approach here considers both the security policy and its implementation. The criteria for a covert channel described above enable one or more bits of information to be passed for each interference event ($\log_2(n)$ bits, where n is the number of possible states that the observer

can differentiate in the shared resource, such as different amounts of delay). When the interference event can be repeated in a cycle, or loop, a stream of data can be transmitted through the channel, although additional synchronization between sender and receiver may be required.

E. DYNAMIC SLICING

Integral to certain covert channels is the notion of data or control dependency. *Slicing* algorithms are used as a means of tracing such dependencies between variables and statements processed during program execution, traditionally for program debugging purposes (Korel and Rilling, 1997). Slicing algorithms generate an executable subset of a program, creating a subprogram whose behavior is the same as the original with respect to some variable. They allow one to isolate the behavior of, and dependencies acting upon, that variable.

Slicing algorithms are categorized as either *dynamic* or *static*, depending on whether they take into account dependencies derived during one particular program execution path (dynamic), or for all possible execution paths (static). Dynamic slicing techniques generally analyze only the narrow portion of the code representing a single execution path.

Since slicing techniques have been shown to be useful in tracking data and control dependencies, they can also provide a means of detecting potential overt flaws based on dependencies. As an example, consider the following code snippet:

```
(s1) Read_dev (SysHigh, v3);
(s2) Read_dev (SysHigh, v4);
(s3) if v3 >5 then
(s4)    v1 := 0;
(s5) else if v4 = 5 then
(s6)         v1 := 1;
(s7)   else v1 := -1;
(s8) v2 := v1;
```

In the example above, it is clear that `v2` depends on `v1`, based on the assignment in (s8). Static slicing shows that `v2`can depend on both `v3` (s3) and `v4` (s5), since there

is a dependency from each of these to `v1`. With dynamic slicing, however, not all execution paths will result in the same control dependencies. When the conditional expression in (s3) evaluates to true, the final value of `v2` depends on `v3` but not on `v4`, since the else-block statements in (s5 – s7) are never executed.

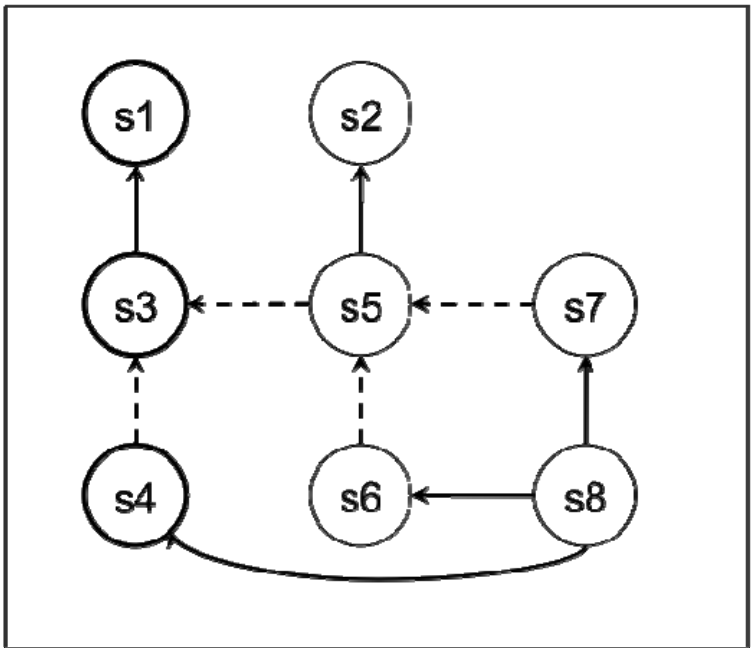

Figure 6. Program Dependency Graph for Code Snippet (after Agrawal and Horgan, 1990).

Figure 6 depicts the program dependency graph created by the execution of the code snippet above. In the graph, the solid edges reflect direct data dependencies, and the dashed edges reflect control dependencies. The bold nodes depict a dynamic slice with respect to the assignment to variable v2 (s8) when the conditional in (s3) has evaluated to true, in which case the assignment depends on statements (s4), (s3), and (s1). Access labels of variables can be used to determine potential security violations, based on the dependencies between these variables. For a finite number of paths within a given scope of analysis, our framework performs static analysis of the DM by using dynamic slicing to discard previous states that could not have contributed to an overt flaw, thus a complete result is obtained without having to maintain a history of all preceding states.

F. INFORMATION FLOW AND CONTROL DEPENDENCIES

With respect to information flows between objects in an MLS system, one category of information flow error occurs when high sensitivity objects indirectly affect information flow between objects at the same (lower) sensitivity level. Such errors are based on unintended *control flow dependencies* (Denning, 1976). For example, during a program execution, information being written from one variable to another, both of the same classification level, might take place within a control structure that depends on a higher classified variable, such as an *if-else* block. In this case, the higher classified *if-else* control variable will create a implicit dependency over the information flow taking place within the control structure, even though this flow is between two lower classified variables. Our approach is capable of detecting this type of overt flaw caused by control flow dependencies. We consider this to be overt, as the program must be processing on behalf of a *High* or *Low* subject, and in either case, reading the *High* control variable and writing to the *Low* variable should not ever be done on behalf of a single-level subject.

The approach used here for discovering flaws based on control dependencies employs dynamic slicing analysis. Dependencies within a program are identified by examining the chain of statements preceding a value assignment with respect to the sensitivity labels of the variables in these statements. If the context of a previous statement includes variables that are higher than the destination, then there is an overt flaw.

The code snippet below provides an illustration of a control flow dependency that constitutes an overt flaw. In the example, constant `value1` is written out to a *SysLow* external device (s3), depending on the *SysHigh* value read into variable `v1` (s1).

```
(s1) Read_dev (SysHigh, v1);
(s2) if v1 > 0 then
(s3)    Write_dev (SysLow, 1);
```

The Write_dev operation in (s3) depends on a *SysHigh* source (`v1`) in the *if*-block (s2), therefore an information flow implicitly exists from `v1` (at *SysHigh*) to the *SysLow* device. Sabelfeld and Myers (2003) described information flows such as these, based on control dependencies, as *implicit flows*.

G. TRUSTED SUBJECTS

Users in a multilevel secure (MLS) environment are assigned a clearance level based on the relative level of trust placed in them by security administrators. A user is allowed to log into a system at any level that is at or below (*dominated by*) his assigned clearance and a *session* at that level is created. Subjects that act on behalf of a given user are labeled with an access class that is at the same level as the user's session. A subject is allowed to read information (*objects*) whose sensitivity level is up to the subject's sensitivity level (*access class*), and write to objects at or above its sensitivity level.

In contrast to this, a *trusted subject* is one that is allowed to read and write within a range of access classes (Lunt *et al.*, 1990), which limits the authority of the trusted subject to "read-up" and "write-down" (Bell and LaPadula, 1973). Lunt *et al.* described that MLS systems with trusted subjects defined this way do not require a separate access control lattice or special rules specifically for their actions. As a result, a trusted subject does not need to be given a privilege to bypass or violate the security rules.

Since trusted subjects are allowed to interact with (read and write) information across access classes, they must be trusted not to abuse these special privileges. The existence of trusted subjects is generally required for certain services provided in MLS systems, such as login, information downgrading, and data backup utilities across multiple access levels. MLS system administration may also require a trusted subject to interact with and manage regular user accounts and information across multiple access levels (Thomas and Sandhu, 1996). Such actions represent a good target for trusted subject implementation, however the design principle (Levin *et al.*, 2007) that trusted subjects should be small and minimized within an MLS system is not always observed.

According to Steffan and Clow (1996), with respect to security policies, a trusted subject should not move data between sensitivity levels, other than in constrained, explicitly defined ways. The specification of a trusted subject must explicitly define how it can do this. Levin *et al.,* pointed out that trusted subjects do not violate the general policy in place, but their behavior must be a defined part of a policy. Such a policy for trusted subjects, referred to as a "relaxed MLS policy," must be integrated with the

general MLS policy, such that the resultant union of the two can allow trusted subjects to effectively operate, while ensuring that non-trusted subjects cannot conduct malicious activity. In a "downgrader" role, for example, a trusted subject may essentially change the label of information from *high* to *low* by reading information from a *SysHigh* object and moving it into another *SysLow* object.

Trusted subjects can be defined by their behaviors in an MLS system. Steffan and Clow (1996) described examples of trusted subject actions, including the ability to process information across multiple access control levels to view (read) a highly sensitive document in order to comment (write) on its contents at a lower level, and the ability to change the sensitivity label contents of a document file. In the latter case, they describe that a trusted subject may *regrade* a classified document, temporarily overriding the tranquility principle that a subject's or object's label will not change while being referenced (Bell and LaPadula, 1973). Although some (Steffan and Clow, 1996) would allow trusted subjects to relabel objects, this dissertation research maintains the tranquility of object labels, abstracting the idea of downgrading information by changing variable labels from the viewpoint of, that is, internal to the trusted subject. Allowing movement of information within a range of access classes represents the trusted subject actions we model in our DM approach.

H. SUMMARY

In this chapter, we have introduced several information assurance concepts germane to the research presented here. The next chapter will expand these discussions by presenting work in several areas related to this dissertation. Some of the work presented is classic in nature and provides a foundation for this dissertation research; in other cases, we present more recent work and contrast it to our research.

III. REVIEW OF RELATED WORK

A. INTRODUCTION

Previous work related to this dissertation research is described below. Discussions include information flow tracing, covert channels and their analysis, control dependencies which may cause overt flaw in a program, system trusted subject implementations, and dynamic security policies.

B. INFO FLOW TRACING AND COVERT CHANNEL ANALYSIS

Related work in modeling secure information flow and in covert channel analysis is described below. We have extended previous work by integrating a language for formally specifying an implementation with a framework for expressing security policies, particularly with respect to covert channel rules and control dependency flaws.

Graham-Cumming and Sanders (1991) described system refinement from abstract to concrete representation, with respect to security, where they defined security solely in terms of noninterference. They used the *unwinding theorem* (Goguen and Meseguer, 1984; Haigh and Young, 1987) to describe refinement such that noninterference between users in an abstract specification of the system could be preserved through more concrete representations of the system, however the results of this work were limited to noninterference. In contrast, our work describes implementation of a proof-of-concept prototype system, where enforcement of a range of security properties is possible through the use of security assertions that are explicitly checked during Alloy program analysis. These abstract security properties are formalized through refinement as security rules (Alloy assertions) in the base program representation of a target program. The semantics of information flow are represented in the DM by the definition of access label ordering, and through the compiler-generated transition predicate, unique to a particular target program.

Volpano *et al.* (1996) furthered the language-based flow analysis work by defining a linguistic type system for secure flow, and rigorously proving the soundness of the core language with respect to noninterference. Well-typed programs are then

guaranteed to be noninterfering, and thus secure by this definition. Sabelfeld and Myers (2003) summarized subsequent work in their survey on language-based information flow systems.

Other work in using sound type systems for secure information flow has focused on areas such as: encryption and decryption of information, where flows from plaintext (*high secrecy*) information to ciphertext (*low secrecy*) information must be addressed in light of noninterference rules that would seem to prevent such interaction (Laud, 2003; Smith and Alpizar, 2006); *probabilistic noninterference*, where probability distributions are used to determine a likelihood of noninterference from *high* to *low* variables, primarily for multi-threaded processes where scheduling is nondeterministic (Volpano and Smith, 1999; Sabelfeld and Sands, 2000; Smith, 2006); and *type inference*, in which the flow of information is automatically determined based on semantic analysis (Volpano and Smith, 1997; Simonet, 2003; Deng and Smith, 2006). Eventually, Smith and Thober (2007) enhanced the linguistic type system model of secure information flow such that sensitivity labels need to be assigned only at I/O boundaries, while the labels of variables and constants, as well as data information flow through a program's execution, are automatically derived relative to the I/O (device) labels.

In contrast, our work implements the DM-Compiler, which similarly tracks the flow of data based on the input device label, but with no requirement to annotate the code in any other way. Our work differs from the linguistic type system approach in that, rather than constructing a type-safe language with which to write secure programs, we apply abstract interpretation to the analysis of programs in order to detect potential problems and otherwise demonstrate their security with respect to select security properties. Our approach is based on exhaustive information flow tracing of all execution paths in a program, to a certain length (determined by the model scope of Alloy). This tracing is applied to both overt and covert channel static analysis using dynamic slicing techniques, where appropriate, such that read-up as well as violations of noninterference (von Oheimb, 2004) are detected. Additionally, we provide a compiler to generate a

formal specification of a program. Although it yet lacks a formal soundness proof, the DM-Compiler enables generation of formal logic that can be automatically analyzed (using the DM) for secure information flows.

Other covert channel research has also focused on information flow analysis, using the principles of Kemmerer's SRM (1983). For example, the concept of *network* covert channel analysis introduced detection methods based on in-depth IP packet analysis as a way to differentiate covert channels from legitimate network traffic (Padlipsky *et al.*, 1978). Approaches such as these could potentially be incorporated into the DM security rule assertions, as methods for detecting covert channels in base programs within this domain.

C. TRUSTED SUBJECT IMPLEMENTATION

In his early work in trusted subject implementation, Wilson (1989) developed a framework for running a trusted multi-level database management system (DBMS), referred to as a "trusted subject," to be hosted on any trusted operating system. This work established a layered policy, with a general policy for the trusted computing base (TCB) layer of the operating system, and a separate policy for the DBMS TCB layer. The goal of this approach was to ensure no illicit disclosure of sensitive DB information (secrecy), and to prevent illicit alteration of DB stored data (integrity). Their premise was that, for a DBMS hosted on a known secure operating system, only the DBMS TCB layer must be subjected to security analysis to ensure that it meets all access control requirements. This concept provided "portability" of the DBMS trusted subject, and negated the time and expense of testing every system on which the DBMS may be targeted. Further, only the DBMS TCB layer need be checked for security when it is operated on a new secure operating system. This work did not appear to outline a traditional concrete security policy for trusted subjects, and only used them in the context of a trusted DBMS. While the Wilson paper allowed modification of object tranquility as a valid action for trusted subjects, we preserve object tranquility by allowing trusted subjects to only modify variable labels during program execution.

Landauer *et al.* (1989) introduced a formal model for managing trusted processes by defining a state machine whose state space can be locked, or isolated, in order to allow privileged actions to overlap, modeling the interleaving of trusted processes. The authors described a trusted process as possessing special privileges to alter operating system kernel access control decisions, or other security critical operations. They divided these privileges into four basic types of trusted process actions which could be taken by trusted users:

1. Change level of data;
2. Perform some integrity-critical functions;
3. Perform service on behalf of non-trusted client (kernel access); or,
4. Export processor information to some non-trusted process.

Additionally, they categorized three general security properties for trusted processes, as components of a trusted security policy (Landauer *et al.*, 1989):

1. Downgrade only at discretion of some privileged subject
2. Execute integrity-critical commands correctly
3. Execute all commands such that resultant state does not violate noninterference (covert channel prevention)

This paper provided an in-depth mathematical analysis of the security policy derived from trusted process principles, and is valuable as a source of background discussion on security policy issues for trusted subjects.

Steffan and Clow (1996) defined a set of trusted process classes, to identify their relative privileged status. These classes correspond to combinations of override privileges in the areas of Tranquility (labels), MAC (content) and DAC (privileges). As the class numbers increase, so do the privileges granted, and the risk associated with using a trusted process in that class. In contrast to this paper, our work characterizes trusted subjects without violating tranquility of object labels.

Thomas and Sandhu (1996) presented three architectures for trusted object-oriented databases, based on: a kernel, a replicated DBMS, and a trusted subject architecture. The last of these was the focus of their paper. Their architecture was

composed of a *session manager*, which was trusted and running across multiple access control layers; several *message managers*, which were untrusted and operated within a single access control layer; and read/write *service requests* to the DB from a client. The trusted session manager can always maintain a global snapshot of the system for a given session, across all access control layers, to allow it to coordinate message requests and scheduling. Clearly, the session manager must be a trusted subject for this architecture to maintain security of messages across layers. As Wilson did in his paper (1989), this work describes the implementation of a trusted subject architecture to support a DBMS. They provided a thorough analysis of how the session manager (trusted subject) could manage messages within the system with respect to security, as well as proofs of both noninterference and confidentiality of the session manager. However, the paper did not appear to focus on security policies for trusted subjects, or how separate policies could effectively coexist.

Levin *et al.* (2006) discussed trusted subject actions within a security kernel architecture. With respect to the principle of least privilege (Saltzer and Schroeder, 1975), they described how a trusted subject in a downgrader role must be able to perform only the minimum required operations, namely, downgrading of labels in this case. Other operations such as "dirty word search" (DWS) of a document for specific words or phrases prior to downgrade, must be handled by other untrusted system processes to prevent unintended or malicious consequences. They defined a framework for performing filtering and downgrade of information, separating tasks between users and processes, both untrusted and trusted. We believe our model is aligned with this thinking, when one considers that if our trusted subject acts as a downgrader, the DM can reflect a separate untrusted process in the target program that performs DWS. We generalize this concept by allowing the trusted subject to modify variable labels based on content or label information. In our model, the DWS might represent examination of a highly classified document for specific references to some classified topic, with subsequent removal of these references prior to downgrading the document. Alternately,

the DWS could represent filtering of a document by its creation date, where downgrading of the document will occur only if this information is older than some predetermined date.

D. DYNAMIC SLICING FOR SECURITY TRACING

Previous work in implementing dynamic slicing algorithms for security property tracing has included development of a tool for finding privacy violations in networked environments, targeting spyware in networked applications (Kruger *et al.*, 2004). The approach uses dynamic slicing techniques to trace program execution, to search for data dependencies that might illuminate privacy violations. When such dependencies are found, they are specified using an event description language to capture event parameters, values, etc. Their goal is to use these parameterized events as abstract inputs for an event sequence language, as a means of generating a security policy. Based on the observed privacy violations, a policy is written that will prevent the specific events that caused the violations. Our suggested approach differs in its goal of analyzing a target program for adherence to a specific policy, as opposed to some generated policy.

E. DOMAIN-SPECIFIC MODELING IN SECURITY

Research has been ongoing in applying domain-specific modeling (DSM) to computer security applications, for example in modeling particular domains such as sensor networks, using DSM principles. Our investigation has not found, however, research in specifically using a security DSM framework for security analysis and verification of programs, such as is the case in this dissertation research. Examples of recent work in this area are discussed below.

Hanna *et al.* (2008) developed Slede, a framework for modeling and verifying sensor network protocols. Their approach uses the *nesC* language for specifying network embedded systems, and the Spin model checker for verifying network security protocols modeled using *nesC*. While their work has parallels to ours, it differs significantly in that its focus is on verification of security protocols, as opposed to verifying programs for adherence to a security policy. It also differs in that it is targeted specifically to sensor networks and not program implementations in general.

Basin et al. (2006), in their *model driven security* approach to system development, point out that software system design models are often disjoint from security models, and their integration is not well understood or supported. Just as automated synthesis of systems from a specification are thought of as a "holy grail" in the software engineering world, the goal of model driven security is automated synthesis of *secure* systems from functional and security specifications. Their approach uses UML to express a security policy, called *secureUML* in their research. While secureUML formalizes access control, with the concepts of subjects, objects, roles (their paper illustrates an RBAC style security policy) and permissions, it does not perform information flow tracing or analysis, as our approach does, and cannot perform covert channel analysis, which we consider a vital aspect of our Security DM approach.

The Security Assertion Markup Language (SAML) was developed by the World Wide Web Consortium (W3C, 2008) as a variant of the more general Extensible Markup Language (XML) to specify security attributes of subjects (users) in a system, such as identity, entitlements, etc. Using established standards for XML Signature and XML Encryption (W3C, 2008), SAML was intended to provide a domain model for security, primarily for business related applications. While SAML can be used to model, for example, digital signatures, message encryption and integrity, and assertions for user authentication, it currently cannot be used to model an access control policy, with assertions for covert channel identification, and could not be applied to our research approach.

The Alloy language has been used to model security *requirements* for secure communications (Chen *et al.*, 2006) where predicates were specified for secure message confidentiality, integrity, authenticity and non-repudiation, as well as numerous "obstacles to security", such as eavesdropping or spoofing. The work was successful in designing a general, reusable model for communication security properties, and differs significantly from the research presented here, which analyzes models of program representations for adherence to a specified security policy.

F. DYNAMIC SECURITY POLICY DEVELOPMENT

Because programs typically interact with an external environment, all aspects of that environment cannot be predicted at compile time. A dynamically secure system must provide mechanisms for observing the environment during runtime, and allow for security updates of the system in order to adapt to that environment (Zheng and Myers, 2004). A static policy cannot provide mechanisms for real-time updates in applications where a change in environment might necessitate immediate enforcement of an updated policy; such a situation requires a specifically dynamic security policy that can adapt to changes in the environment surrounding the system.

A *dynamic security policy* can be defined as a program consisting of a set of guards and actions that provide the logic to modify a system's implementation in order to change its operational parameters, and includes the necessary guards to enforce good behavior and prevent misuse of the system (Naldurg *et al.*, 2002). Dynamic policies provide adaptive access control measures that can be responsive to time (temporal aspects of the system), events (emergent situations or unanticipated actions in the environment), perceived security risk, and operational needs.

The goal of *Quality of Service* (QoS), the ability of a distributed system to provide sufficient and timely service to meet the desires of each of its users, is expanded to Quality of Security Service (QoSS), with the vision of making security a constructive tool for network management, rather than a performance inhibitor (Irvine and Levin, 2000; Levin *et al.*, 2006). A QoSS framework becomes a directing factor in the implementation of a dynamic policy, employing variant mechanisms that make security decisions based on changing network operating conditions, always working to maximize benefit to its users (Irvine and Levin, 2000). Research in this area has included study in the development of adaptive policies for management of databases (Ray, 2005), tools for design and verification of dynamic policies (Janicke *et al.*, 2005), and the manner in which an adaptive policy should be implemented.

The DoD and National Security Agency (NSA) are currently developing the Global Information Grid (GIG) (NSA GIG, 2004), which will provide the military with

greater net-centric communications capability and flexibility by leveraging "information technology and innovative network-centric concepts of operations to develop increasingly capable joint forces." The GIG will address limitations and inefficiencies inherent in the military's traditional specialized "stove-pipe" communications systems by providing the ability to share information across networks of differing classification levels, and across coalition networks.

Among the GIG's information assurance goals are high assurance authentication, multi-level security, and development of flexible, dynamic security policies. To that end, the GIG program has introduced a new type of security policy known as *Risk-Adaptive Access Control* (RAdAC). The RAdAC policy represents a confluence of MAC and DAC, enforcing a *need-to-know* policy, with SAC (Situational Access Control), which enforces a more adaptive *need-to-share* policy. RAdAC is described by a general rule that "users may access information they are cleared and permitted to access, or when the situational need is severe enough, as long as the risk of doing so is within tolerable limits," (NSA GIG, 2004). The RAdAC model would provide the flexibility to meet *need-to-share* requirements, weighing potential risk against operational need, making decisions to grant access based on:

- security risk in granting access (primarily a function of the user, the object being accessed, the environment in which they exist, and historical allowed accesses)
- operational necessity for access
- any policy for a balance between the two in conflicting situations

G. SUMMARY

In this chapter, we have presented classic and recent related work in secure information flow tracing, and specially the relationship of this to covert channel analysis and noninterference properties. We have described research in the trusted subject concept, and how others have chosen to implement trusted subject behaviors in a secure system. Finally, we have described the DoD's goal of finding solutions to implementing dynamic security policies, particularly for military secure systems.

The next chapters will introduce the approach presented in our work, beginning with the Implementation Modeling Language (IML).

IV. IMPLEMENTATION MODELING LANGUAGE

A. INTRODUCTION

The Implementation Modeling Language (IML) defines a specialized language that presents some of the basic capabilities and constructs, with respect to security, of high-level programming languages. The current IML enables the specification of relatively simple programs written in some common programming language, such as Java or C++. While future iterations of IML might handle more advanced language features such as concurrency, inheritance, etc., this initial language description was motivated by a requirement to represent essential security information flow properties in target program implementations, balanced by the desire to limit complexity during experimentation.

B. IML SYNTAX

The following describes syntax and statement constructs of the IML.

1. Lexical Concepts

A variable name is an identifier distinct from IML keywords and Alloy keywords. No variable declarations are required.

The only assumption about values stored in variables is that they can be compared for equality and inequality (<, =, >, <=, >= operators) with other variables, or with constants. Variables can hold integer constants, but the value of a variable can be interpreted also as a time value (see `GetClock` below). Constants are represented by integers: `-1`, `0`, `1`, etc.

Statement constructs provided in IML include capabilities for assignment to a variable, reading to and writing from a variable, accessing an I/O device's flags and a system clock, and basic control structures. Semicolons separate statements in IML.

2. Assignment

Assignment statements propagate access labels from the right-hand side to the left-hand side of the statement. For the current model, constants have a *Low* secrecy access label by default.

variable **:=** ***variable*****;**

variable **:=** ***constant*****;**

The IML enables trusted subject behavior by providing a special *trusted assignment* statement. This statement allows trusted subjects to modify the labels of internal variables ("regarding"), while respecting the tranquility of external object labels. The trusted assignment allows filtering of variable values based on existing content and/or label. This filtering is analogous to a "dirty word search" of a document prior to downgrading its classification level, to ensure that certain sensitive words are first filtered from the document.

The trusted assignment statement allows a trusted subject to assign a value to a *destination* variable, with an explicitly defined security label. When an IML base program is translated, it is under the context that only a trusted subject may perform trusted assignment. The trusted assignment syntax follows:

Assign ***destination*** **from** ***source*** **as** ***alt_source*****;**

In this operation, the *destination* variable takes on a new data value (*destination_value'*) from the *source* variable, however it does not automatically take the *source* label as would normally be the case for an assignment statement in IML. Instead, *destination* is explicitly assigned a new label (*destination_label'*) based on the *source* and *alt_source* labels, as determined by a filter function that is automatically invoked with each trusted assignment. In trusted assignment, *source* can be either a variable or constant, and *alt_source* can be either a variable (in which case the access label currently assigned to the value stored in this variable is used) or an explicitly defined access label.

The new content and access label of the *destination* variable (*destination_value'* and *destination_label'*, respectively) are defined by the Alloy function **tsFilter** in the DM Invariant Model (further discussed in section 4.1.1), as follows:

(*destination_value'*, *destination_label'*) =

`tsFilter` ((*destination_value*, *destination_label*),
(*source_value*, *source_label*),
(*alt_source_value*, *alt_source_label*))

This function specifies the behavior of trusted subjects in our model, and examples are described in detail in Chapter VI.

3. Device Input/Output Statements

The IML statements `Read_dev` and `Write_dev` abstract the input from and output to an external device at a specific access level. We make the simplifying assumption that there are three external devices, each at *high*, *medium* and *low* secrecy levels, respectively; the operation label (*SysHigh, SysMid* or *SysLow*) indicates which of these devices is being read to or written from. For a `Read_dev` statement, the variable is assigned the label of the device that is read from; for a `Write_dev` statement, source may be either a variable or a constant.

`Read_dev (`*`label, variable`*`);`
`Write_dev (`*`label, source`*`);`

4. File Random Access

The IML abstracts the concept of random access to an indexed file, where *(key, value)* pairs are used to store and retrieve information in a finite-sized repository. This conceptual repository, referred to as a *direct file*, can be thought of as a database or memory file and (for this model) is represented as a single-level store (there is no distinction between persistent and volatile memory).

All subjects in the base program can access a single instance of the direct file, according to their access label. Initially, all direct file slots have a *SysLow* access label, and can be written to by any subject. Once a subject has stored a value into a keyed slot using the `PutDirectFile` statement, that slot retains the label of the subject. Subsequently, another (or the same) subject may read from this direct file slot using the `GetDirectFile` statement, only if the subject's label dominates that of the key slot.

A given key slot can be overwritten an unlimited number of times by a subject with a higher- or lower-labeled value, so the label of a given slot may change over time.

The direct file has a limited number of keyed slots, all of which have empty keys and values at the start of program execution, and a given slot's key value is determined when it is first assigned a key/value pair. The direct file tracks the number of slots that have been assigned a key, zero at the start of execution and incremented by one whenever a key slot in the direct file is written to for the first time. The direct file capacity equates to the number of key slots that can be allocated in the direct file.

When a `PutDirectFile` is executed for a given key for the first time, an available key slot is allocated, the data is stored in the direct file, and a global *Success* flag is set to 1; otherwise, if no key slot is available, the *Success* flag is set to 0, and no data is stored. When all available slots have been allocated, the direct file is considered filled, and a global *Full* flag is set to 1. The *Success* and *Full* flags are global state variables maintained by the execution environment, and are internal resources that would not be directly accessible in a high-level language. Their values could be inferred, however, based on system errors seen by the user, and we abstract such system errors in the IML by allowing direct examination of the flags in a base program.

The following statements are provided in the IML for storing and retrieving values to/from the direct file. The label indicates the level of the subject performing the operation; the key and source may be either variables or constants:

```
GetDirectFile (label, key, variable);
PutDirectFile (label, key, source);
```

5. System Clock

This statement stores the current clock value to a variable:

```
GetClock (variable);
```

We model only the time taken by file and external device accesses during `Read_dev/Write_dev` and `Get/PutDirectFile` operations. These statements may cause the CPU, or some other resource, to be busy such that some action visible to another subject is delayed with respect to a reference clock (for simplicity, we model one time source – the system clock).

The clock value can be compared with other constants and variables, using the `Before` operator:

```
(var1 Before var2)
```

6. Program Control Statements

A conditional expression is constructed from variables, constants, flags, and operators =, >, <, >=, <=, `Before`, `not`, `and`, `or`. A *statement* may be any single statement or block of statements (a sequence of statements is enclosed by braces). Two forms of control statements are provided:

```
if condition then statement [else statement];
while condition do statement;
```

In the `if-then-else` statement, the `else` block is optional. The `while-do` control statement repeats its body as long as the *condition* holds true.

The following statement signifies termination of a base program:

```
Stop;
```

C. SUMMARY

This chapter introduced the Implementation Modeling Language (IML). This specialized language, developed as an integral part of the Domain Model (DM) concept, provides a way to represent high-level language programs in a common modeling format, to enable automated static analysis of a representation the program's execution. The IML provides all of the functionality necessary for program analysis of security properties, particularly with regard to information flow analysis.

We next present the Security DM approach, which enables automated static analysis of programs to verify them for adherence to a specifically defined security policy.

THIS PAGE INTENTIONALLY LEFT BLANK

V. THE SECURITY DM APPROACH

A. INTRODUCTION

This chapter presents the Domain Model (DM) approach to security verification. It provides discussions of the overall structure of the DM framework, including the *Invariant* and *Implementation Models*, and the compilation process during which the DM-Compiler is used to generate a complete DM, in Alloy notation, from a base program written in the IML.

B. DM STRUCTURE

An overview of the Security Domain Model (DM) approach to program security verification is depicted in Figure 7. The DM includes the definition of program state and transitions between states, as well as security rules, specified as Alloy assertions, representing the generic policy a program must conform to. The DM is composed of an invariant and a variable section, derived from the security rules and a target implementation, respectively.

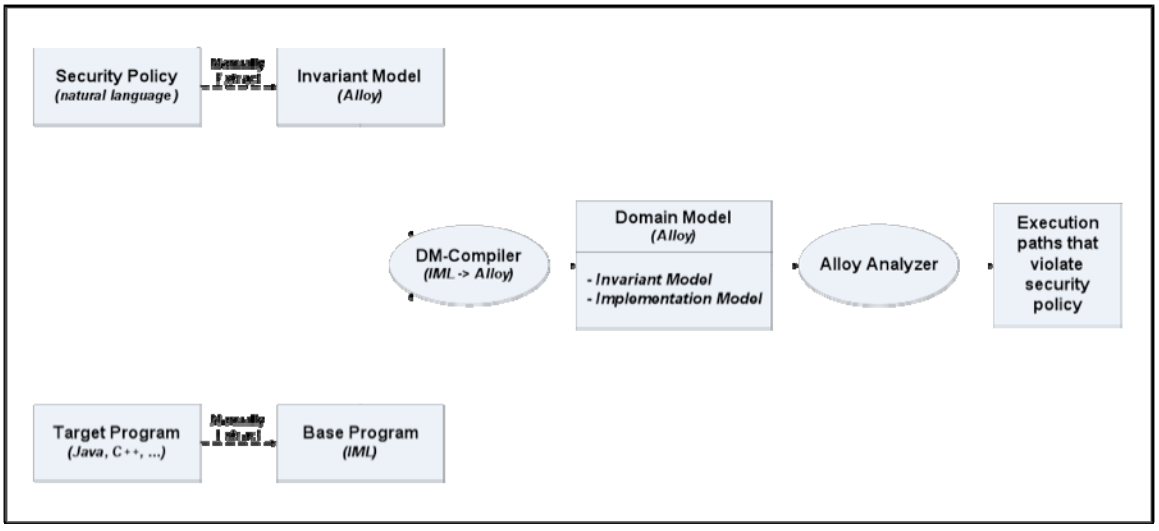

Figure 7. Domain Model approach to system security verification.

While there are numerous model checker tools currently available, we chose to use the Alloy specification language primarily because of its ability to represent program

language abstractions simply and explore their semantics with a well-integrated analysis tool. As Jackson (2006) points out, referring to his approach as "lightweight formal methods," Alloy models can be easily created and initially tested early in the development process, and then incrementally expanded. He states that the goal of Alloy was to "obtain the benefits of traditional formal methods at lower cost, without requiring a big initial investment," presumably in time and effort.

As do traditional model checkers, Alloy deals with finite models though it handles them very differently. Model checkers typically build Kripke structures to represent the states and transitions of a program execution. Such finite model structures have limits not easily adjusted by the user during analysis. The Alloy Analyzer tool, however, affords the ability to easily increase the depth of analysis for models as they are developed and expanded. For our approach, Alloy and its Analyzer provide an ideally suited tool for creating and analyzing target program abstractions.

As previously explained, a *base program* is an abstraction of a *target program* implementation, and is written in IML notation. By analyzing a model of the program rather than actual program code, security verification can focus on elements of information flow such as I/O, access labels, direct file access, and timing (clock), while ignoring other program details not pertinent to such analysis.

In the current prototype, translation of the base program from an implementation is a manual step. Developing a separate compiler to translate a high-level language program to IML is a difficult task, beyond the scope of this work. The possibility must be considered that covert channels existing in the original program implementation may be lost in the IML representation, and for now we depend on the knowledge of the manual translator to avoid this problem.

Security rules, written as Alloy assertions, are derived from the security policy. Such policies are typically written in natural language, and extraction of security rules is a manual step in our approach. As currently implemented, the DM defines security rules, which have as their basis the Bell and LaPadula (1973) security model, meaning that information flows from *High* to *Low* secrecy levels are not allowed.

After the base program and Invariant Model with security rules are defined, the DM-Compiler compiles the base program from IML into state transition predicates, written in Alloy notation, creating the DM Implementation Model. The DM-Compiler combines this with the Invariant Model to complete the DM. The approach uses the Alloy Analyzer tool (Alloy, 2008) for automated verification of the security rules, defined in the DM as Alloy assertions, to find execution paths within the DM that might violate the security policy or create covert channels. In essence, it creates an interpreter for the specific base program, modeled by the DM.

1. Invariant Model

The Invariant Model specifies the conceptual framework of the DM with the Alloy specification language. This section describes statement types and structure, program execution state, direct file structure, and clock signature.

In the Alloy language, all atomic structures are modeled as sets and relations. Sets are represented as unary relations; scalars are simply singleton sets. A set or relation declaration can be constrained using several keywords indicating multiplicity: `one` restricts sets to exactly one instance of a type; while `lone` restricts them to either zero or one instance; and `none` refers to the empty set. The `all` quantifier must hold for all instances of a type, and the `disj` quantifier specifies variables that are necessarily disjoint from one another.

Alloy provides standard logical operators, for example negation (`!`), conjunction (`&&`), disjunction (`||`), implication (`=>`), and bi-implication (`<=>`). Pairs `(type->type)` represent binary relations, and '`+`' is the set union operator. The override operator '`++`' examines two sets of pairs and overwrites the pair in the first set with the second whenever the first elements of the pairs match. The '`^`' operator represents transitive closure for binary relations.

The signature (`sig`) construct in Alloy, roughly synonymous with the class declaration in object-oriented programming languages, defines a set of atoms (elements), and any relations between them. Signatures with the `abstract` qualifier cannot have

their own instances, and are used only to derive other signatures. For further details on the Alloy language, see the website at (Alloy, 2008).

The signatures below describe program state, the initial state, and structures for variables and values, which are extended in the DM-Compiler generated Implementation Model (discussed in next Section). The `Policy` signature defines a partial ordering between security access labels, representing a dominance relationship between the labels (see Figure 8). For illustration purposes, the model defines an Alloy enumerated type, `AccessLabel` with access labels *SysLow, SysMid* and *SysHigh*. The `Policy` signature defines an ordering `ord` as the transitive closure between the three labels, as well as the identity, or reflexive, relationship for each of the labels.

```
enum AccessLabel { SysHigh, SysMid, SysLow }

one sig Policy {
  ord: AccessLabel -> AccessLabel
}
{ord = ^( (SysLow -> SysMid)
        + (SysMid -> SysHigh) )
        + (iden & (AccessLabel -> AccessLabel) )
}
```

Figure 8. Alloy enumerated type for `AccessLabel`, and signature for the DM `Policy` element

The `Statement` abstract signature (see Figure 9) captures a single instance of a given statement. For I/O (Read_dev/Write_dev) and direct file access statements, the signature defines statement `type`, `destination`, `source`, `key` (for direct file only) and `subject_label` attributes. The `subject_label` specifies the security label of the calling subject for a particular statement; this label represents the access label of the device, in the case of I/O statements. For assignment statements, only `source` and `destination` attributes are defined. For conditional statements, the `source` attribute

defines the set of control variables used in an `if-then-else` or `while-do` statement. For GetClock statements, only the `destination` attribute is defined, while the Stop statement defines no attributes.

```
sig Statement {
  type:          Stmt_type,
  destination:   lone Variable,
  source:        set Variable + Value,
  source_label:  lone (AccessLabel + Variable),
  key:           lone (Variable + Value),
  subject_label: lone AccessLabel
}
```

Figure 9. Alloy signature for the DM `Statement` element

The `Stmt_type` enumerated type (see Figure 10) defines the different statement types that can be used in a base program representation of a target program.

```
enum Stmt_type {
  Assign,
  Condition,
  Read_dev,
  Write_dev,
  GetDirectFile,
  PutDirectFile,
  GetClock,
  Stop
}
```

Figure 10. Alloy enumerated type for the DM `Stmt_type` element

The `DirectFile` signature (see Figure 11) defines key/value pairs (`keyContent`) for each of its storage slots, and the current access label (`keyLabel`) for each key slot value. The latter is used to track the label of the current value to ensure that flows are valid during subsequent accesses of the value using `GetDirectFile`

statements. The element `last_written` stores the label of the last subject that wrote to the direct file, and is used when checking for potential covert storage channels (discussed in detail later). The signature also defines the direct file `max_slots` size (set to 2 for modeling purposes); note that Alloy provides the predefined type `Int` to represent sets of integer atoms. Also, `full` and `success` are used as internal resource system flags (essentially Booleans), as previously described in Chapter IV.

```
sig DirectFile {
  keyContent:   Value -> lone Value,
  keyLabel:     Value -> lone AccessLabel,
  last_written: lone AccessLabel,
  full:         (const0 + const1),
  success:      (const0 + const1),
  max_slots:    Int
}
{ max_slots = 2
}
```

Figure 11. Alloy signature for the DM `DirectFile` element

```
sig Time {}

one sig Clock {
  before:      Time -> Time,
  long_before: Time -> Time
}
{ long_before in before &&all t1: Time, t2: Time - t1 |
  ((t1->t2) in before <=> t2 in TO/nexts[t1]) &&
  ((t1->t2) in long_before <=> some t3: Time |
    (t3 in before[t1] && t3 in before.t2))
}
```

Figure 12. Alloy signatures for the DM `Time` and `Clock` elements

The `Clock` signature (see Figure 12) provides an abstraction for program execution time. The signature defines the concept of some event occurring at some time before another event (`before` relation), which enables testing for the relative timing of events during base program analysis. In this implementation, '`TO`' is defined as a `Time` ordering instance, using the Alloy library utility for ordering. The `nexts` function returns a set of all next values in an ordering – in this case the next `Time` values after the one in question. For example, the code below checks whether `t2` is contained within the set of time values that occur after `t1`.

```
sig State {
  stmt:              Statement,
  vars:              Variable -> one (Value + Time),
  access_label:      Variable -> one AccessLabel,
  direct_file:       DirectFile,
  current_clock:     Time,
  prev_state:        lone State,
  err_msg:           lone Error,
  influenced_by:     Variable -> State,
  last_cond_checked: set State,
}
{
  ( err_msg = InfoFlow_error <=>
    not consistent_with_FlowPolicy [this] ) &&
  ( err_msg = Overt_flaw_detected <=>
    dependency_flaw_found[this] ) &&
  ( err_msg = Storage_channel_detected <=>
    storage_channel_found[this] ) &&
  ( err_msg = Timing_channel_detected <=>
    timing_channel_found[this] )
}
```

Figure 13. Alloy signature for the DM `State` element

The `State` signature (see Figure 13) captures the current state of the system, and the next base program statement (`stmt`) to be executed during static analysis. Its elements include the type of statement to be executed, the current table of variable values,

the access label for each value stored in `vars` (for information flow tracing), and a snapshot of the current direct file; the flags `full` and `success` are contained within the `direct_file` attribute. This signature also includes the `current_clock` value, the previous state leading to the current state, and `last_cond_checked`, which identifies a set of conditionals within which the current statement may be nested, enabling dependencies from those conditionals to be propagated.

The `influenced_by` attribute is used for tracking control flow dependencies, and is at the heart of the dynamic slicing algorithm used in this approach. It stores, for each `source` variable in the current state, all of the previous states that have influenced that variable. This attribute enables the Alloy Analyzer to narrow its focus in examining previous states, thus reducing the search space necessary in determining control dependencies. By storing variable/state pairs, we can enable the Analyzer to examine all variable access labels from previous influencing states.

The `State` signature also defines specific error conditions, referred to as `err_msg` in the signature. These errors represent security violations which might occur during static analysis, and provide positive feedback that the Alloy Analyzer has discovered an assertion counterexample and a potential violation. For modeling purposes the `State` signature currently defines errors for illicit information flows (flows which violate the `Policy` signature partial orderings), overt control dependency flaws, and covert channels, to include both storage and timing channels, as defined in the Alloy enumerated type `Error` (see Figure 14).

```
enum Error {
  InfoFlow_error,
  Overt_flaw_detected,
  Storage_channel_detected,
  Timing_channel_detected
}
```

Figure 14. Alloy enumerated type for the DM `Error` element

In order to provide a starting state for static analysis of a base program, the DM Invariant Model defines an initialization signature, called `InitialState`, which sets appropriate values for the various elements of the `State` signature (see Figure 15).

```
one sig InitialState extends State {}
{
  vars                = (Variable -> const0)
  access_label        = (Variable -> SysLow)
  stmt                = S1
  direct_file.full    = const0
  direct_file.success = const1
  current_clock       = TO/first[]
  prev_state          = none
  err_msg             = none
  last_cond_checked   = none
  no influenced_by
  no direct_file.keyContent
  no direct_file.keyLabel
}
```

Figure 15. Alloy signature for the DM `InitialState` element

When analyzing a base program, the Alloy Analyzer performs an exhaustive search of all execution paths up to a defined length, referred to as the *scope,* specifying the upper limit of the size of the models considered. In fact, it performs symbolic execution of all base program paths with length up to the defined scope. In the DM-Compiler, the scope is generated heuristically based on the total number of statements in a base program. For example, the resultant scope for a given base program will encompass all of its control statements and I/O statements, and will be one more than the number of statements in the program to ensure that the initial state is included in the analysis. The heuristic addresses while-loop statements in a base program to a limited extent, by increasing the scope to allow for a single iteration.

The DM-Compiler scope heuristic ensures that all execution paths for deterministic programs will be scrutinized, however, it cannot be assured of generating execution paths for non-deterministic programs. For example, the Analyzer cannot generate all possible execution paths for a program with a while-loop structure of indeterminate length, thus a scope cannot be determined that will ensure all possible paths are scrutinized.

The Invariant Model also includes the definition of security rules that must be enforced by the DM security policy. These rules are specified as Alloy assertions, and will be described further in Chapter VI.

2. Implementation Model

The DM Implementation Model is generated by the DM-Compiler from a base program, and specifies the base program's semantics in terms of statement signatures and state transitions. Example base programs, and their resultant compiled Alloy models, are presented in Section 5.

From the base program, the DM-Compiler generates `Variable` and `Value` signatures, representing variable names and constant values used in the base program, respectively. The `Variable` signatures reflect the variables defined in the base program; similarly, the number and value of constants defined in the `Value` signature depend on the number and value of unique constants explicitly present in the base program (the constant 0 will always be added by default for initial variable values). To represent the state space, additional constants may be needed to fill the intervals between explicitly defined constants. The DM-Compiler defines an Alloy signature which establishes a simple *less-than* relationship between the constant values, thus enabling the comparison of values for equality and inequality from within the base program. Figure 16 shows example `Variable` and `Value` signatures generated by the DM-Compiler.

```
one sig x1, x2, t1
  extends Variable {}
one sig const_minus_1, const0, const1, const2, const3
  extends Value {}
one sig LT {
  lt:  Value -> Value }
{ lt = ^(
      ( const_minus_1 -> const0)
    + (  const0 -> const1)
    + (  const1 -> const2)
    + (  const2 -> const3) )
}
```

Figure 16. Example Alloy signatures for `Variable` and `Value` elements

C. DM-COMPILER

The DM-Compiler compiles each base program statement into a separate Alloy signature, based on the type of statement and associated variables and constants used. Elements of the `Statement` signature not needed for a particular statement type are not initialized. The base program in Figure 17 shows a simple signature sequence, translated into IML, for an assignment statement (s2) nested within a conditional statement (s1).

```
(s1) if ( x1 < 0 ) then
(s2)   x2 := x1;
(s3) Stop;
```

Figure 17. Sample Base Program Statements, in IML Syntax

From the sample base program above, the DM-Compiler would generate the sequence of Alloy Statement signatures shown in Figure 18.

```
one sig s1 extends Statement {}
{ type = Condition
  source = x1
  destination = none
  key = none
}
one sig s2 extends Statement {}
{ type = Assign
  source = x1
  destination = x2
  key = none
}
one sig s3 extends Statement {}
{ type = Stop
  source = none
  destination = none
  key = none
}
```

Figure 18. DM-Compiler Generated Alloy Signatures for Sample Base Program

From these statement signatures, the DM-Compiler generates a *transition predicate* representing the state transition trace for the base program execution. The approach used to derive the transition predicate is based on compilation of the base program statements, using the RIGAL compiler construction language (Auguston, 1990; Auguston, 1991). During the compilation process, each base program statement is translated into a set of Alloy statements that represent the transition from the present execution state to the next state, following execution of the statement in question. The transition predicate defines a *frame condition* (Borgida *et al.*, 1995) for each statement, capturing the semantics of the base program by specifying all possible sequences of statement executions for the base program. The transition predicate also implements dependency tracking within the execution path. Although we refer generally to the

transition “predicate,” we represent this structure using an Alloy `fact` rather than a `pred` (predicate), because a `pred` only holds when invoked, while a `fact` is assumed to always hold.

The remainder of this section shows a representation of the state transition predicate derived by the DM-Compiler for the base program in Figure 17. Note that for each statement, `pre` represents a state before the statement (`stmt`) has been executed, and `post` represents the state after statement execution.

```
fact trans {
  all post: State - InitialState |some pre: State |
```

For the conditional statement (s1), since no variable value assignments are made, the variable table, access labels, direct file (including system flags), clock time value, and `influence_by` table remain the same after execution:

```
(pre.stmt = s1 &&
  (post.vars = pre.vars &&
   post.access_label = pre.access_label &&
   post.direct_file = pre.direct_file &&
   post.current_clock = pre.current_clock &&
   post.influenced_by = pre.influenced_by &&
```

The `last_cond_checked` attribute is calculated to include all previous states currently in `last_cond_checked` (excluding the current state, s1), plus the `pre` state itself, in order to set the context of statements within the conditional:

```
post.last_cond_checked =
     {cond: pre.last_cond_checked |cond.stmt != s1 } + pre &&
```

Based on the outcome of the conditional check, the next statement to execute is set to either the “then” branch (s2), or the “else” branch (s3) statement:

```
    (((pre.vars[x1]-> const0) in LT.lt) => post.stmt = s2
       else post.stmt = s3)
    )
&& post.prev_state = pre
) ||
```

In the assignment statement (s2), the access label and value for the target variable (`x2`) are set to those of the source variable (`x1`):

```
(pre.stmt = s2 &&
  (post.vars = pre.vars
     ++ (x2 -> pre.vars[x1]) &&
   post.access_label = pre.access_label
     ++ (x2-> pre.access_label[x1]) &&
   post.stmt = s3 &&
```

The direct file (including system flags), clock value, and `last_cond_checked` attribute all remain the same after execution of an assignment:

```
   post.direct_file = pre.direct_file &&
   post.current_clock = pre.current_clock &&
   post.last_cond_checked = pre.last_cond_checked &&
```

The `influenced_by` attribute is calculated based on the source variable dependencies. Recall that `influenced_by` is declared within the `State` sig as the relation (*Variable->State*), which is a set of pairings from variables to states. Alloy treats sets and subsets the same when defining relations, thus the pairing of a variable to a set of states (Variable->{State}), shown below, denotes a set of pairings from that variable to each of the states ({Variable->State}). Jackson (2006, pp. 36-37) provides a discussion of Alloy's treatment of sets as relations.

In calculating `influenced_by`, first all previously recorded dependencies (other than those for `x2`, the destination variable) are included:

```
post.influenced_by =
   {v: Variable, s: State | (v -> s) in pre.influenced_by && v != x2}
```

Second, dependencies for `x1`, the current assignment statement source variable, are added as dependencies for `x2`:

```
   + (x2 -> pre.influenced_by [x1])
```

Next, from the current set of states defined in `last_cond_checked`, those whose scope this assignment falls within are included. This captures dependencies from any conditional within which the current statement may be nested; in this case base program statement (s1):

```
   + (x2-> {cond: pre.last_cond_checked | cond.stmt = s1}
)
```

Finally, when an assignment statement is nested within a conditional statement, dependencies from the `source` variables participating in the conditional must be included:

```
    + (x2 -> State.{cond: pre.last_cond_checked,
        infl: cond.influenced_by [cond.stmt.source] | cond.stmt = s1})
    ) && post.prev_state = pre
) ||
```

The transition predicate concludes with the Stop statement (s3). Since execution terminates when this point is reached there is no need to assign values for the resultant (`post`) state, other than setting the previous state for tracing continuity:

```
(pre.stmt = s3 &&
 post.prev_state = pre
)}
```

D. SUMMARY

This chapter has described the Security DM approach to static analysis and verification of a program representation for adherence to a security policy – specifically,rules associated with a security policy. Specifically, we described the structure of the Security DM, and how it is generated based on a base program representation of a target program, and a specific security policy.

In the next chapter, we present several base program examples for which the Security DM approach is used to conduct static analysis for the presence of specific security violations and adherence of the base program to a security policy.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. EXAMPLE DM IMPLEMENTATIONS

A. INTRODUCTION

This chapter presents examples of program security vulnerabilities, discoverable using the DM approach. The examples illustrate security rules – defined using Alloy notation – for discovering direct information flow errors, indirect control dependency flaws, and covert channels, based on regular and trusted subject actions. For each example, a base program written in IML is presented to demonstrate a particular security violation, and how the domain model approach can be used to find it.

The success of these examples shows the possibility of conducting automatic analysis and verification of target programs. For specific program instantiations, we have demonstrated the characterization of several classes of security flaws and successfully analyzed example programs automatically for presence of these flaws. These results indicate a direction for future research, to represent broader abstractions for automatically detecting entire classes of security flaws in a target program.

Regarding the covert channel examples presented, each describes the transmission of one bit of information, for conceptual purposes. More complex real-world examples would involve such concepts as looping, synchronization, etc., to provide exploitable covert channels with a stream of bits.

The complete Alloy models for these and other examples can be found on the dissertation research website at http://cisr.nps.edu/projects/sdm.html.

B. EXAMPLE PROGRAMS

1. Overt Control Dependency Flaw

The first example illustrates an overt flaw based on a control flow dependency. This example shows an exploitation scenario that culminates with an IML `Write_dev` operation, where the variables written to the external device have been influenced by values at a higher level than that of the device itself.

The Alloy predicate in Figure 19 examines each execution state, and evaluates as true whenever the state (`current`) is the result of a `Write_dev` statement, and the value to be written out was `influenced_by` some previous state (`pre`) that had access to a variable with a higher `access_label` than that of `current` state, that is, the flow from the previous State to the current State is from *SysHigh* to *SysLow*.

```
pred dependency_flaw_found [current: State] {
let stm = current.stmt, pre = current.influenced_by[stm.source]| {
  stm.type = Write_dev  &&
  stm.source in Variable &&
  not ((pre.access_label[pre.stmt.source] -> stm.subject_label)
     in Policy.ord)
  }
}
```

Figure 19. Alloy Predicate to Discover Overt Control Dependency Flaw

The following base program illustrates an example of this flaw. Initially, a *SysHigh* value is read into variable `x1` (s1). Based on the value of `x1` (s2), new variable `x2` is assigned either '0' (s3) or '1' (s4). Variable `x2` is then written to a *SysLow* device (s5).

The violation occurs when `x2` is written to a *SysLow* device, because its value has been potentially influenced by a *SysHigh* value, specifically `x1` when it was accessed in (s2).

```
(s1) Read_dev (SysHigh, x1);
(s2) if x1 = 0 then
(s3)    x2:= 0;
(s4) else x2:= 1;
(s5) Write_dev (SysLow, x2);
(s6) Stop;
```

The Alloy Analyzer detects this situation, and correctly reports an overt flaw by tracing the control flow through statements (s1)(s2)(s3)(s5).

2. Timing Covert Channel Resulting from Exploitation of System Clock

The second scenario describes a covert timing channel that occurs when a *SysLow* subject executes two `GetClock` statements, and between them a *SysHigh* subject prevents the *SysLow* subject from executing, through execution of a Read_dev/Write_dev or direct file operation (`rw` state in the text). Thus, when the *SysLow* subject next runs, it can examine the clock to detect this interference with its access to the CPU; these channels are thus often called *CPU channels*. The Alloy assertion in Figure 20 detects this potential covert timing channel, utilizing the system `Clock` element.

```
pred timing_channel_found [gc2: State] {
some disj rw, gc1: State | {
  (gc2 -> rw) in State_order.st_after &&
  (rw -> gc1) in State_order.st_after &&
  gc1.stmt.type = GetClock &&
  gc2.stmt.type = GetClock &&
  rw.stmt.type in (Read_dev + Write_dev +
                   PutDirectFile + GetDirectFile) &&
  gc1.stmt.subject_label = gc2.stmt.subject_label &&
  not ((rw.stmt.subject_label -> gc2.stmt.subject_label)
     in Policy.ord)
  }
}
```

Figure 20. Alloy Predicate to Discover Timing Covert Channel

The base program below illustrates this timing channel. A *SysHigh* value is initially read into variable `x1` (s1). A *SysLow* subject then stores the current clock value in `t1` (s2). Based on a check of `x1` (s3), its value is stored into the direct file at key slot 1 (s4). The *SysLow* subject again examines the clock, and stores its value into `t2` (s5).

```
(s1) Read_dev (SysHigh, x1);
(s2) GetClock (SysLow, t1);
(s3) if x1 < 0 then
(s4)   PutDirectFile (SysHigh, 1, x1);
(s5) GetClock (SysLow, t2);
```

At this point (subsequent to execution of statement s5) an interference event has occurred, which can be exploited as a timing covert channel by the *SysLow* subject, and the Alloy Analyzer detects the violation, tracing execution flow through statements (s1)(s2)(s3)(s4)(s5). The crux of this covert channel is that a *SysLow* subject, the covert channel receiver, has been allowed to observe (by examining the clock) a change in some internal resource (the CPU busy state), which was indirectly affected by the actions of a *SysHigh* subject, the covert channel sender. The remaining statements illustrate how the *SysLow* subject compares the two clock values (s6) to see whether the *SysHigh* subject has interfered with it through performance of some operation, and writes either a '1' or '0' accordingly (s7 and s8).

```
(s6) if t1 Before t2 then
(s7)   Write_dev (SysLow, 1);
(s8) else Write_dev (SysLow, 0);
(s9) Stop;
```

3. Flow Violation Caused by a Trusted Subject Operation

The third example illustrates a trusted subject regrade operation that, based on allowed trusted subject behavior, leads to an information flow violation. In the example, an attempt is made by a trusted subject to downgrade a destination variable label from *SysHigh* to *SysLow*. Here, trusted subjects are allowed to perform downgrading of information from *SysHigh* to *SysMid*. To support the policy, a `tsFilter` Alloy function is defined in Figure 21 to ensure that any "downward" information flows are allowed only from *SysHigh* to *SysMid*. The function takes as input parameters three `Values` and three `AccessLabels`, specifically, the values and labels of the *destination*, *source* and *alt_source* variables in the Trusted Assignment (see Chapter IV for IML syntax of the trusted assignment statement), and returns as its `result` an instance of Alloy signature `FTuple`, which the DM uses as the new (filtered) `Value` and `AccessLabel` of *destination* (see Figure 21). In essence, the policy for trusted subject behaviors is captured in the semantics of the `tsFilter` function, which may override the normal value and label of the assignment *destination* parameters.

For example purposes, the `tsFilter` function here returns the greater of constant 0 and the *source* `Value` (`source_val`), and the higher of *SysMid* and the *alt_source* `AccessLabel` (`alt_src_label`) when a downgrade is performed; otherwise it returns `alt_src_label` for an upgrade or when the label is not changed. As shown in the example `tsFilter`, it is not necessary to use all of the parameters passed into the function to generate a resulting `FTuple`. Note that a different DM Invariant Model might define a `tsFilter` function that would return different results based on the specific input parameters, and thus define a different security policy for trusted subject behaviors.

```
sig FTuple {
  val:   Value,
  label: AccessLabel
}

fun tsFilter[ dest_val, source_val, alt_src_val      : Value,
              dest_label, source_label, alt_src_label: AccessLabel ]:
  FTuple { { result: FTuple |
  {

//assign result.val to be the greater of source_val and 0
  result.val =
       (((source_val -> const0) in LT.lt)
           => const0
       else source_val)

//assign result.label to be the higher of SysMid and alt_src_label,
//  when downgrade is executed; otherwise assign alt_src_label
  result.label =
       (((dest_label -> alt_src_label) in Policy.ord)
           => alt_src_label
       else SysMid)
  }
} }
```

Figure 21. Alloy Function for Trusted Subject Filter

The base program example below demonstrates a security violation based on the trusted subject filter and security policy. Initially, values are read into two variables with security labels *SysHigh* and *SysMid*, respectively (s1 and s2). A trusted assignment operation is then performed (s3), in which the data value stored in `x2` is copied into variable `x1`, and `x1` is assigned a *SysLow* label. The trusted assignment statement invokes the tsFilter function, which overrides the label assignment, from *SysLow* to *SysMid* (as described above), resulting in *destination* variable `x1` being assigned a higher label (*SysMid*) than was intended (*SysLow*).

```
(s1) Read_dev (SysHigh, x1);
(s2) Read_dev (SysMid, x2);
(s3) Assign x1 from x2 as SysLow; //now x1 has the label SysMid
(s4) Write_dev (SysLow, x1);
(s5) Stop;
```

When the next statement (s4) attempts to write the value held in `x1` to a *SysLow* external device, an illicit flow results since `x1` is labeled as *SysMid*. The Alloy Analyzer detects this situation as a violation of the information flow security predicate in Figure 22, and correctly reports an illicit information flow, tracing execution through statements (s1)(s2)(s3)(s4). The same base program, under a DM Invariant Model with a different policy and filter function, would not necessarily result in this flow violation.

```
pred consistent_with_FlowPolicy [current: State] {
  let stm = current.stmt | {
    (stm.type in (Write_dev + PutDirectFile) &&
     stm.source in Variable )
    => (current.access_label[stm.source] -> stm.subject_label)
       in Policy.ord
  }
}
```

Figure 22. Alloy Predicate to Discover Illicit Information Flow

4. Trusted Subject Dual Violation – Information Flow Violation and Overt Flaw

The fourth example base program illustrates two different security violations that may result from a trusted subject operation. In the program, a successful trusted subject regrade creates an overt control dependency flaw, however when the trusted subject regrade fails to occur, illegal information flow results. For purposes of this example, the security policy and tsFilter function described above apply.

In the base program, values are initially read into three variables, with assigned security labels *SysHigh*, *SysMid* and *SysLow*, respectively (s1 through s3). Depending on the value stored in `x1` (s4), a trusted assignment statement is performed (s5) in which the value of `x1` is modified to the greater of `x2` and 0, and the label of `x1` is downgraded to that of `x3`, *SysMid* in this case. Since a regrade from *SysHigh* to *SysMid* is allowed by the security policy (as reflected in the tsFilter function), `x1` is assigned the *SysMid* label.

```
(s1) Read_dev (SysHigh, x1);
(s2) Read_dev (SysLow, x2);
(s3) Read_dev (SysMid, x3);
(s4) if x1 < 0 then {
(s5)   Assign x1 from x2 as x3;   //now x1 has the label SysMid
(s6)   Write_dev (SysMid, x1); }
(s7) else Write_dev (SysMid, x1);
(s8) Stop;
```

The next statement (s6) attempts to write the value of `x1`, which is now labeled *SysMid*, to a *SysMid* external device. However, since this operation occurs within the if-block, it creates a control dependency from *SysHigh* (`x1` label when it was examined in s4) to *SysMid*, representing an overt access control flaw (in the *SysHigh* context, a write to *SysMid* violates the security policy). Based on the Alloy security rule predicate (see example in section 5.1), the Alloy Analyzer properly detects this violation, tracing execution through statements (s1)(s2)(s3)(s4)(s5)(s6).

An additional violation occurs when the conditional check (s4) evaluates to false, and the else-branch is executed. In this case, an attempt is made to write the value stored in `x1` (still assigned its original *SysHigh* label) to a *SysMid* external device (s7). Since

this represents an overt illegal flow from *SysHigh* to *SysMid*, the Alloy Analyzer properly identifies and reports the error, tracing execution through statements (s1)(s2)(s3)(s4)(s7).

5. Storage Covert Channel Resulting from a Trusted Subject Operation

The final example combines the concepts described in the previous ones by showing how the execution of a trusted assignment could produce a covert storage channel (Levin *et al.*, 2006). This example demonstrates that, even with a consistent security policy for trusted subjects, common data flow violations that are outside the trusted subject's allotted permissions may occur in a base program. Security violations, such as covert channels or information flows in violation of the policy, may be perpetrated by the illicit actions of regular subjects, regardless of the actions of a trusted subject.

The DM formalizes the notion of covert channels with an Alloy security predicate (see Figure 23) to identify a class of covert storage channel vulnerability in a base program execution.

```
pred storage_channel_found [current: State] {
  let stm = current.stmt | {
    stm.type = PutDirectFile &&
    current.direct_file.full = const1 &&
    not (current.direct_file.last_written -> stm.subject_label)
       in Policy.ord
  }
}
```

Figure 23. Alloy Predicate to Discover Storage Covert Channel

In the example base program below, we assume a direct file with a maximum capacity of two records, initially empty. To begin, *SysLow* values are read into variables `x1` and `x2` (s1-s2). A trusted assignment is then performed (s3) in which `x1` is assigned the value of the greater of `x2` and 0 (based on the `tsFilter` function described above), and upgraded to a *SysHigh* label. At this point in execution, the value stored in `x1` will be 0 or greater. Next, the value of `x1` is examined (s4). When this check evaluates to

true, the values of `x1` and `x2` are stored into the direct file by the *SysHigh* sender, resulting in the internal *full* direct file flag being set.

```
(s1)  Read_dev (SysLow, x1);
(s2)  Read_dev (SysLow, x2);
(s3)  Assign x1 from x2 as SysHigh; //now x1 has the label SysHigh
(s4)  if x1 > 1  then {
(s5)    PutDirectFile (SysHigh, 1, x1);
(s6)    PutDirectFile (SysHigh, 2, x2); }
```

Figure 24 graphically depicts the sequence of events that take place during execution of the code sequence above. This execution results in the Direct File being filled by the storage channel sender at *SysHigh* when the values stored in variables `x1` and `x2` are written as *SysHigh* labeled values to direct file key locations 1 and 2, respectively. At this point, the direct file is full.

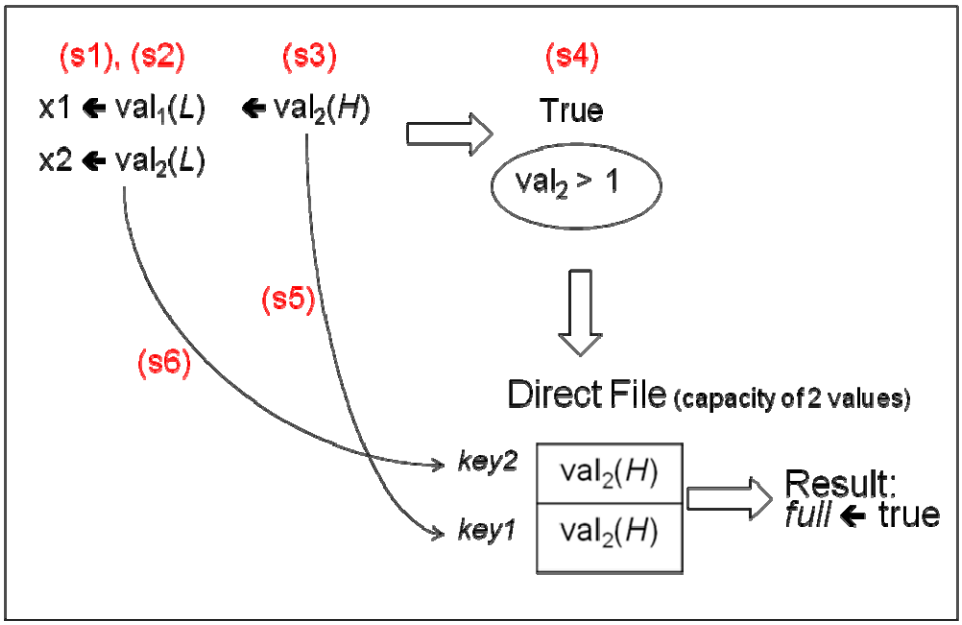

Figure 24. Direct File filled by storage channel *SysHigh* sender

The next sequence of program statements represents execution by a *SysLow* covert channel receiver. When the *SysLow* subject attempts to store a value into the direct file using a new key 3 (s7), the system issues a failure indication since the direct file is *full* (note that in the translation to a base program, the internal system flag returned as an error message to the program, translates to the explicit `full` flag, accessible in

IML as in statement (s8)). Depending on the success or failure of the direct file store (s8), whether or not the covert channel receiver found the direct file to be full, the *SysLow* subject writes a constant '1' or a '0' to an external device (s9 & s10) to complete the storage channel.

```
(s7)  PutDirectFile (SysLow, 3, 1);
(s8)  if full = True then
(s9)    Write_dev (SysLow, 1);
(s10) else Write_dev (SysLow, 0);
(s11) Stop;
```

Because a higher-labeled subject caused the direct file to become full, the Alloy Analyzer detects and reports this violation of the Alloy security predicate, tracing the flow of execution through statements (s1)(s2)(s3)(s4)(s5)(s6)(s7). Although this violation occurred subsequent to a trusted subject action (s3), the storage channel was not precipitated by the trusted assignment, and the illicit actions of two regular subjects at *SysHigh* and *SysLow*, acting in collusion to exploit the direct file, brought about the security violation (a storage channel).

C. TESTING RESULTS

The base program examples presented above were evaluated using Alloy Analyzer version 4.1.8, running under Mac OS X™ 10.5.4 on a 2.5 GHz Intel Core 2 Duo processor, with 2 GB of memory. In test runs, the Alloy Analyzer successfully found valid counterexamples for violations of each security rule assertion described above.

Test results are summarized in Table 1 below. The Analysis Size defines the size of Alloy model instances considered *(scope)* during static analysis; Analysis Time represents *total time (ms),* broken down into *(time to build model, time to analyze and find a counterexample)*:

Security Violation Description	**Analysis Size, *scope***	**Analysis Time, *ms* (build, analyze)**
Overt control dependency flaw	7	688 (640, 48)
Timing covert channel	10	5891 (2771, 3120)
Information flow violation, resulting from trusted subject operation	7	1516 (1277, 239)
Overt control dependency flaw, resulting from trusted subject operation	9	3335 (2290, 1045)
Information flow violation, resulting from trusted subject operation	9	2692 (2236, 456)
Storage covert channel, resulting from trusted subject operation	12	48631 (9852, 38779)

Table 1. Results of Alloy Analysis Testing

With regard to the state explosion dilemma associated with most model checkers, testing was conducted to study the effect of DM base program size on Alloy Analyzer static analysis time. As discussed earlier, the Alloy Analyzer tool relies on the small scope hypothesis to assert that any flaws in a model are found in relatively small instances of that model. With this in mind, static analysis was performed on simple base programs of steadily increasing size, with associated increasing analysis scope. Note that in each example base program, a security flaw was intentionally implemented in order to ensure that a counter-example could be discovered by the Alloy Analyzer.

Figure 25 shows the results of these test runs. The chart graphically depicts how, with increasing base program size (measured as lines of IML code), the time taken for the Alloy Analyzer tool to find a valid counter-example to one of the security assertions increased with exponential growth. This result was not unexpected, given that the Alloy model size for increasing DMs tends to grow exponentially during Alloy Analysis, and even a relatively small scope can equate to a very large search space (Jackson, 2006). With increasing architecture speeds, and efficiencies gained with new Alloy improvements, perhaps the ability to test larger and larger base programs will become possible.

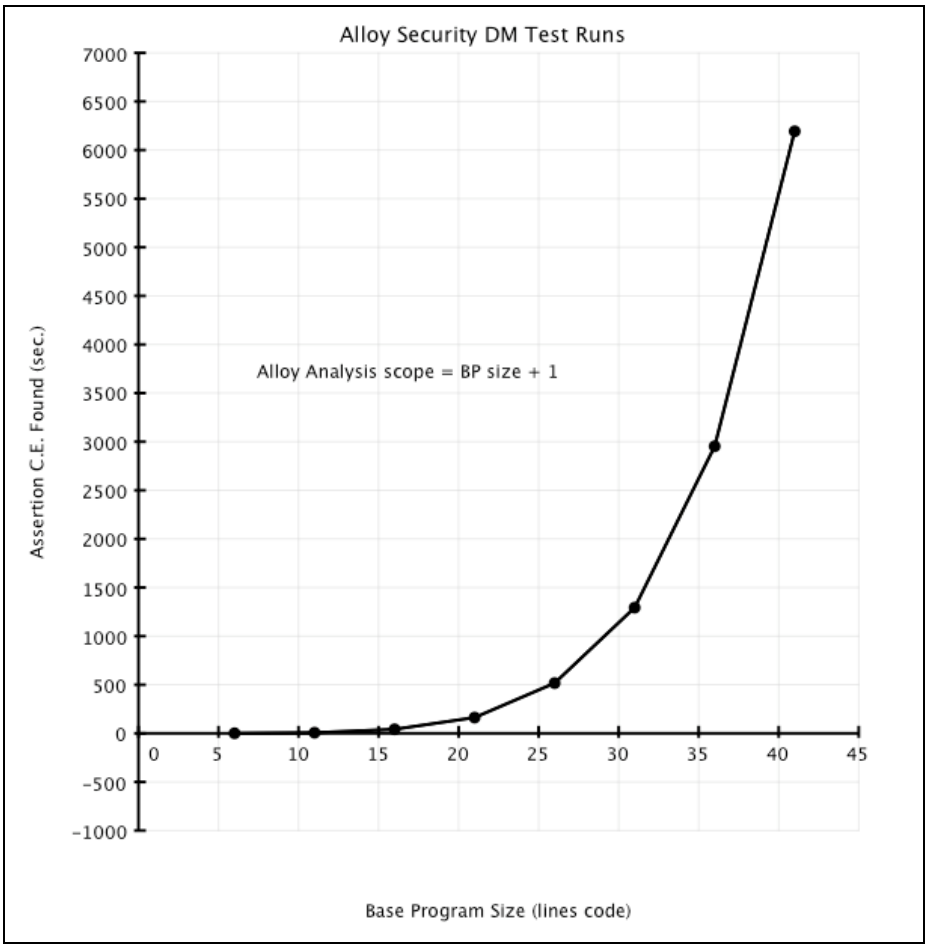

Figure 25. Alloy Analyzer Static Analysis Times for Increasing Base Program Sizes

D. SUMMARY

This chapter has demonstrated the feasibility of the Security DM approach to program verification. We have demonstrated that this approach can be used successfully to perform static analysis of a representation of a target program, and verify its adherence to a security policy abstraction, represented by a set of formalized security rules.

We will next present conclusions from this work, and propose areas for future advances in this research.

VII. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

This dissertation has described the development of a security domain model for representing high-level language programs and security policies, for both regular and trusted subjects. The approach defines a formal Security Domain Model (DM) that facilitates specification of a security policy (security rule assertions) and security vulnerabilities (such as covert channels), and is independent of a particular program implementation. Although encoding and checking static program semantics and properties is not in itself revolutionary, this work is evolutionary in extending previous work in the area of information flow tracking based on a precise, formal definition for overt information flaws and covert channels.

The Security DM framework provides a means of conducting automated static analysis of a program implementation within a finite scope of execution paths. Flow control dependencies and related overt flaws are analyzed using dynamic slicing techniques. Our model has the ability to identify security flaws such as covert channels, as well as overt flaws in a program, which may allow illicit information flow control dependencies. In addition, we provide special trusted behavior in our DM by representing a trusted subject. Our implementation of a relatively small trusted subject is in line with the Reference Monitor Concept principle that a reference validation mechanism "must be small enough to be subject to analysis and tests" to ensure its correctness (Anderson, 1972).

Using the Alloy Analyzer to perform static analysis of a base program, as represented by an Alloy language DM, provides assurance that a counterexample to a security rule will be found when one exists. This is true for analysis of deterministic programs of a finite length, which are assured of terminating. For such programs, a *scope* of analysis is calculated based on the number of statements in the program. This ensures that all program statements, and by extension all execution states, are included in a generated model of the program, thus Alloy Analyzer static analysis results in neither

false negatives nor false positives within the defined scope of analysis. In the case of false negatives, assuming a proper scope of analysis is chosen, every execution path is examined thus if a security violation (as defined by an Alloy assertion) exists, it will be reported. In the case of false positives, a violation will only be reported when the specific conditions defined by a security assertion exist in some base program execution path. Because a reported violation can be investigated in the wake of static analysis, false positives would seem to be less egregious than false negatives.

We have shown through examples that the Security DM approach is able to generate a unique specification for a target program representation (base program) and security policy (security assertions), and automatically identify counterexamples for the security assertions where they exist, to identify security violations in base program representations of target programs. We base our conclusion in part on the small scope hypothesis, and while it is just that – a hypothesis – previous work (Andoni *et al.*, 2002) has demonstrated its validity, and the effectiveness of systematic testing within a small scope (as the Alloy Analyzer performs) as compared to larger scale testing of fewer inputs, for refuting specification assertions.

This dissertation research has introduced several contributions that are integral to the Security DM approach, reiterated below:

1. Implementation Modeling Language (IML)

The IML is a specialized language that supports basic information processing to facilitate static analysis of a representation of high-level language source code, by providing a formalism that captures the essence of imperative programming language paradigms, while ignoring non-essential (for these purposes) elements.

2. Security Domain Model (DM)

The security Domain Model (DM), represented as an Alloy specification, provides a model of a target program behavior, as well as a model for describing security

properties. The DM is a unified representation of a base program representation of the target program, and the intended information flow security policy, including restrictions on both overt and covert information flow.

3. DM-Compiler

A specialized compiler was developed to translate a base program, written in IML, into an *Implementation Model*, and then integrated with the *Invariant Model* to form a complete DM specification to represent the original *target program*.

B. RECOMMENDATIONS FOR FUTURE WORK

In the wake of this dissertation work, a number of areas for further research have been identified. These areas fall generally into two broad categories: formal analysis of the DM and its artifacts, and expansion of the DM to facilitate more complex programming constructs and to allow formalization of more advanced security policies.

1. Correctness of the DM

Currently, we define information flow not in theoretical terms, but in terms of Security DM constructs. For example, we formalize an illicit flow in terms of the access labels defined in a base program statement, and their relationship to labels associated with variables currently in existence at some execution point. Through static analysis the DM discovers violating flows, but how can we be assured that this static analysis is correct? An important area for future research would be to address the correctness of the DM with respect to a formal information flow property, such as noninterference.

In order to demonstrate correctness of the DM, we must identify an information flow security property independent of the Security DM, and demonstrate that whenever static analysis concludes that a program is secure, it is indeed secure in the sense that it exhibits this property. For example, can it be shown that programs identified by the DM as secure indeed satisfy the noninterference (or some other) property for deterministic programs?

2. Formal Analysis of DM Artifacts

Currently, extractions of base programs from target programs, and iteration of security rules from a natural language security policy, are manual steps in our approach. Future work can focus on formally analyzing the semantics of the IML and DM-Compiler to ensure that the artifacts of each (the *base program* and *implementation model*, respectively) are accurate abstractions of the original target implementation.

As Sabelfeld and Myers (2003) pointed out, information flow analysis should take place "as close to the execution code as possible." Formal analysis of this process of extracting the base program from a target program to ensure an absolute correspondence between the two will help achieve the goal of being "as close…as possible" to the target program source code.

3. IML Expansion

The IML currently supports basic information processing using assignment statements, conditional and loop statements, read/write statements, file random access, and access to a system clock. In order to focus specifically on programming constructs that support static analysis for information flow tracing, we essentially ignored more advanced constructs, such as data type, inheritance, polymorphism, etc. To enable modeling of a larger domain of high-level language constructs, and target program examples, the IML can be extended greatly to capture these and other programming language constructs.

4. Dynamic Security Policies

Future work could expand the DM to enable dynamic security policies (Levin *et al.*, 2006). This concept would allow the DM to support, for example, a sequence of polices during program execution, and support the ability of a system to adapt to a dynamically changing security environment (NSA GIG, 2004). This could be extended by adding functionality for multiple trusted subjects, each potentially operating under different trusted policy rules. By defining multiple filter functions within a DM Invariant

Model, and modifying the IML syntax to support this, the model could represent separate trusted subjects, each governed by a different policy as defined by its own filter function.

Traditional security models are aimed at ensuring a computer system is statically secure by showing that, given a proven secure state of the system (typically the initial state) and a secure transition from that state to another, the resultant state must be secure. By extension, if all state transitions are secure, the entire system must be secure. Under a dynamic security policy, however, a transition could take place which might not be considered secure. For example, consider an uncleared user in the field who requires immediate emergency access to a highly sensitive piece of information in order to accomplish his mission. Once that sensitive object has been exposed to the uncleared user, an unsecure transition has taken place leaving the system in an unsecure state, thus the system is no longer secure.

However, in a dynamic policy sense, perhaps the system is still perfectly secure despite the transition; the question is - how can this be modeled? Beyond that, can the system be returned to a secure state without unjustifiably modifying the sensitivity level of the user or information, that is, without raising the clearance level of the user or declassifying the information, when neither might be appropriate? Alternatively, could the policy be defined to automatically declassify the information after such an access, or after a specified time period? If our example was modified to one where a malicious uncleared user had accessed the same piece of sensitive data, clearly this should represent an unsecure situation, but how does the system recognize this difference? These are some of the questions that must be addressed in the definition of a dynamic security policy.

5. Networked Analysis

Padlipsky *et al.* (1978) introduced the concept of *network* covert channel analysis and introduced detection methods based on in-depth IP packet analysis as a way to differentiate covert channels from legitimate network traffic. Approaches such as these could potentially be incorporated into the DM security rule assertions, as methods for detecting covert channels in base programs within this domain.

6. Model-Driven Software Development

Future work in this research can focus on tailoring our approach toward the model-driven software design process. It is understood that automation of the software development cycle, such that resulting software systems fully conform to the Common Criteria evaluation requirements, is not a trivial effort. We have focused specifically on the Implementation Representation and Security Objectives stages of development (Common Criteria, 2003), the base program and security policy assertions respectively, devising an automated way to verify that the former adheres to the latter. A framework to automate the actual production of these artifacts would be an ideal goal for future development in this work.

LIST OF REFERENCES

The Alloy Analyzer, http://alloy.mit.edu/. Accessed October 2008.

Agrawal, H., and Horgan, J. (1990). Dynamic program slicing. *ACM SIGPLAN Notices,* 25(6), 246-256.

Anderson, J. P. (1972). *Computer Security Technology Planning Study*, Technical Report ESD-TR-73-51, Volumes I and II. NTIS document AD-758 206, ESD/AFSC, Hanscom Air Force Base, Bedford, MA, USA.

Andoni, A., Daniliuc, D., Khurshid, S., and Marinov, D. (2002). Evaluating the "small scope hypothesis." *Proceedings of the 29th ACM Symposium on the Principles of Programming Languages* (POPL'02).

Auguston M. (1990). Programming language RIGAL as a compiler writing tool, *ACM SIGPLAN Notices*, 25(12), 61-69.

Auguston M. (1991). *RIGAL - a programming language for compiler writing*. Lecture Notes in Computer Science, Springer Verlag, vol. 502, 529-564.

Basin, D., Doser, J., & Lodderstedt, T. (2006). Model driven security: From uml models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology, 15*(1), 39-91.

Bell, D., LaPadula, L. (1973). Secure Computer Systems: Mathematical Foundations and Model, *MITRE Report*. The MITRE Corp.

Bell, D., LaPadula, L. (1976). Secure Computer Systems: Unified Exposition and Multics Interpretation, *MITRE Report*. The MITRE Corp.

Biba, K.J. (1977). Integrity Considerations for Secure Computer Systems. MITRE Corp. Technical Report ESD-TR-76-372.

Borgida, A., Mylopoulos, J., and Reiter, R. (1995). On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10), 785-798. IEEE Press.

Chen, C., Grisham, P., Khurshid, S., and Perry, D. (2006). Design and validation of a general security model with the alloy analyzer. *Proceedings of the ACM SIGSOFT First Alloy Workshop*, 38-47.

Chou, S. (2005). An RBAC-Based access control model for object-oriented systems offering dynamic aspect features. *IEICE Transactions on Information Systems*, 2143-2147.

Clark, D., and Wilson, D. (1987). A Comparison of Commercial and Military Computer Security Policies. *Proceedings of the IEEE Symposium on Security and Privacy (S&P'87),* 184-194.

Common Criteria for Information Technology Security Evaluation, version 3.1. Document number CCMB-2006-09-001. September 2006.

Corin, R., Durante, A., Etalle, S., and Hartel, P. (2003). A trace logic for local security properties, *International Workshop on Software Verification and Validation (SVV'03)*. Mumbai, India.

Deng, Z., and Smith, G. (2006). Type inference and informative error reporting for secure information flow. *Proceedings of the 44th ACM Southeast Conference* (pp. 543-548). Melbourne, Florida.

Denning, D. E., and Denning, P. J. (1977). Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 504-512. ACM Press.

Denning, D. (1976). A Lattice Model of Secure Information Flow. *Communications of the ACM 19(5)*, 236-243.

Department of Defense Trusted Computer Security Evaluation Criteria, DOD 5200.28-STD, National Computer Security Center, December 1985.

Ferraiolo, D., and Kuhn, R. (1992). Role-Based Access Control. *Proceedings of the 15th National Computer Security Conference*, 554-563.

Ge, X., Polack, F., and Laleau R. (2004). Secure databases: an analysis of Clark-Wilson model in a database environment. Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04), 7-11.

Gligor, V. (1993). *A guide to understanding covert channel analysis of trusted systems*. Technical Rep. NCSC-TG-030, National Computer Security Center, Ft. Meade, MD, USA.

Goguen, J., and Meseguer, J. (1982). Security Policies and Security Models. *Proceedings of the IEEE Symposium on Security and Privacy 1982*, 11-20.

Graham, G., and Denning P. (1972). Protection: Principles and Practices. *Proceedings of the 1972 AFIPS Spring Joint Computer Conference*, 417-429.

Graham-Cumming, J., and Sanders, J.W. (1991). On the refinement of noninterference. *Proceedings of the Computer Security Foundations Workshop IV* (pp.35-42).

Haigh, J.T., and Young, W.D. (1987). Extending the noninterference version of MLS for SAT. *IEEE Transactions on Software Engineering*, SE-13(2), 141-150.

Hanna, Y., Rajan, H., and Zhang, W. (2008). A domain-sopecific verification framework for sensor network security protocol implementations. *Proceedings of the ACM Conference on Wireless Network Security* (WiSec'08).

Harrison, M., Ruzzo, W., and Ullman, J. (1976). Protection in Operating Systems. *Communications of the ACM, 19(8)*, 461-471.

Irvine, C., and Levin, T. (2000). Toward Quality of Security Service in a Resource Management System Benefit Function. *Proceedings of the 2000 Heterogeneous Computing Workshop*, 133-139.

Irvine, C., and Levin, T. (2000). Quality of Security Service. *Proceedings of the New Security Paradigms Workshop*, 18-22.

Irvine, C., and Levin, T. (2001). Data Integrity Limitations in Highly Secure Systems. *Proceedings of the International System Security Engineering Association Conference*.

Irvine, C. and Levin, T. (2002). A cautionary note regarding the data integrity capacity of certain secure systems. In M. Gertz, E. Guldentops, L. Strous (Eds.), *Integrity, Internal Control and Security in Information Systems*, (3-25). Norwell, MA, USA: Kluwer Academic Publishers.

Irvine, C. E., Levin, T. E., Nguyen, T. D., and Dinolt, G. W. (2004). The Trusted Computing Exemplar Project, *Proceedings of the 2004 IEEE Systems, Man and Cybernetics Information Assurance Workshop* (pp. 109-115). West Point, NY, USA.

Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA, USA, London, England: MIT Press.

Janicke, H., Siewe, F., Jones, K., Cau, A., and Zedan H. (2005). Analysis and Run-time Verification of Dynamic Security Policies. *AAMAS'05 Workshop on Defence Applications of Multi-Agent Systems.*

Karger, P. and Wray, J. (1991). Storage channels in disk arm optimization. *Proceedings of the IEEE Symposium on Security and Privacy*, 52-63.

Klein, G., and Huuck, R. (2005). High Assurance System Software, *Proceedings of the 10th Australian Workshop on Safety Related Programmable Systems (SCS'05)*. Sydney, Australia: Australian Computer Society, Inc.

Kruger, L., Wang, H., and Jha, S. (2004). *Towards discovering and containing privacy violations in software* (Technical Report No. 1515). Madison, WI, USA: University of Wisconsin-Madison.

Lampson, B. (1971). Protection. *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, 437-443.

Landauer, J., Redmond, T., and Benzel, T. (1989). Formal policies for trusted processes, *Proceedings of the Second IEEE Computer Security Foundations Workshop (CSFW'89)* (pp. 31-40). Franconia, New Hampshire, USA: IEEE Computer Society.

Landwehr, C. (1981). Formal Models for Computer Security. *ACM Computing Surveys, 13(3)*, 247-278.

Laud, P. (2003). Handling encryption in analyses for secure information flow. *Proceedings 12th European Symposium on Programming, ESOP* (pp. 159-173).

Levin, T., Irvine, C., and Nguyen, T. (2004). *A Least Privilege Model of Static Separation Kernels* (Report No. NPS-CS-05-003). Monterey, CA: Naval Postgraduate School.

Levin, T., Irvine, C., and Nguyen, T. (2006). Least Privilege in Separation Kernels. *Proceedings of the 2006 International Conference on Security and Cryptography*, 355-362.

Levin, T., Irvine, C., and Spyropoulou, E. (2006). *Quality of security service: Adaptive security*. Handbook of Information Security (H. Bidgoli, ed.), vol. 3, 1016–1025. Hoboken, NJ: John Wiley and Sons.

Levin, T., Irvine, C., Weissman, C., and Nguyen, T. (2007). Analysis of three multilevel security architectures. *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, 37-46. ACM Press, New York, NY.

Lunt, T., Denning, D., Schell, R., Heckman, M., and Shockley, W. (1990). The seaview security model. *IEEE Transactions on Software Engineering*, 16(6), 593-607.

McLean, J. (1990). Security models and information flow. *Proceedings of the IEEE Symposium of Security and Privacy*, 180-189. IEEE Computer Society Press.

McLean, J. (1994). Security Models. Excerpt from Encyclopedia of Software Engineering (ed. John Marciniak), Wiley Press.

McLean, J. (1996). A general theory of composition for a class of "probabilistic" properties. *IEEE Transactions on Software Engineering*, 22(1), 53-67. IEEE Press

Naldurg, P., Campbell, R. H., and Mickunas, M. D. (2002). Developing Dynamic Security Policies. *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)*, 204-215.

National Institute of Standards and Technology. (2007). *Source Code Security Analysis Tool Functional Specification*, Version 1.0 (NIST Special Publication No. 500-268). Gaithersburg, MD: Black, P., Kass, M., and Koo, M.

National Security Agency IA Directorate. (2004). *Global Information Grid Information Assurance Reference Capability/Technology Roadmap*, Version 1.0.

National Security Agency. (2007). *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, Version 1.03.

Neumann, G. (1998). Consideration of the Chinese Wall and Clark-Wilson Security Policy in the Internet Environment. *Seminar Paper, Department of Information Systems and Software Techniques, University of Essen, Germany*.

Nguyen, T. D., Levin, T. E., and Irvine, C. E. (2005). TCX Project: High Assurance for Secure Embedded Systems, *11th IEEE Real-Time Embedded Technology & Applications Symposium (RTAS'05)* (presented as Work-in-Progress). San Francisco, CA, USA: IEEE Computer Society.

Padlipsky, M., Snow, D., and Karger, P. (1978). *Limitations of end-to-end encryption in secure computer networks*. MITRE Technical Report, MTR-3592, Vol. I, May 1978 (ESD TR 78-158, DTIC AD A059221).

Ray, I. (2005). Applying Semantic Knowledge to Real-Time Update of Access Control Policies. *IEEE Transactions on Knowledge and Data Engineering,* 17(6), 844-858.

Ryan, P., McLean, J., Millen, J., and Gligor, V. (2001). Noninterference, who needs it? *Proceedings of the 14th IEEE Computer Security Foundations Worskshop (CSFW'01)*, 237-238.

Sabelfeld, A., and Myers. A. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 5-19. IEEE Press.

Sabelfeld, A., and Sands, D. (2000). Probabilistic noninterference for multi-threaded programs. *Proceedings of the IEEE Computer Security Foundations Workshop* (200-214).

Saltzer, J. and Schroeder, M. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1278-1308.

Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. (1996). Role-Based Access Control Models. *IEEE Computer, 29(2)*, 38-47.

Schaefer, M., Gold, B., Linde, R., & Scheid, J. (1977). Program confinement in KVM/370. *Proceedings of the 1977 Annual ACM Conference*, 404-410. ACM Press.

Schell, R., Tao, T., and Heckman, M. (1985). Designing the GEMSOS Security Kernel for Security and Performance. *Proceedings of the 8th National Computer Security Conference*, 108 - 119.

Schell, R. (2001). Information Security: Science, Pseudoscience, and Flying Pigs, *IEEE 17th Annual Computer Security Application Conference* (ACSAC'01), 205-218. IEEE Computer Society.

Schellhorn, G., Reif, W., Schairer, A., Karger, P. A., Austel, V., and Toll, D. (2000). Verification of a Formal Security Model for Multiapplicative Smart Cards. *Proceedings of the 6th European Symposium on Research in Computer Security* (ESORICS'00), 17-36.

Security Domain Model Project website, http://cisr.nps.edu/projects/sdm.html. Accessed October 2008.

Shaffer, A., Auguston, M., Irvine, C. and Levin, T. (2007). Toward a security domain model for static analysis and verification of information systems. *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, 160-171. Montreal, Canada.

Shaffer, A., Auguston, M., Irvine, C. and Levin, T. (2008). A Security Domain Model to Assess Software for Exploitable Covert Channels. *Proceedings of the ACM SIGPLAN Third Workshop on Programming Languages and Analysis for Security* (PLAS'08), 45-56. Tucson, Arizona.

Shaffer, A., Auguston, M., Irvine, C. and Levin, T. (2008). A Security Domain Model for Implementing Trusted Subject Behaviors. *Proceedings of Workshop on Modeling and Security* (MODELS'08), Toulouse, France, September 28, 2008, http://ceur-ws.org/Vol-413.

Shaffer, A., Auguston, M., Irvine, C. and Levin, T. (2008). A Security Domain Model for Static Analysis and Verification of Software Programs. *Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering* (SEKE'08), 673-678. San Francisco, California.

Shaffer, A., Auguston, M., Irvine, C. and Levin, T. (2008). *A Security Domain Model for Static Analysis and Verification of Software Systems*. Manuscript submitted for publication.

Shockley, W. (1988). Implementing The Clark/Wilson Integrity Policy Using Current Technology. *Proceedings of the 11th National Computer Security Conference*, 29-37.

Simonet, V. (2003). Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'03)*, vol 2895 (pp. 283-302). Beijing, China: Springer-Verlag.

Smith, G. (2006). Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security* 14(6), 591-623.

Smith, G., and Alpizar, R. (2006). Secure information flow with random assignment and encryption. *Proceedings of the 4th ACM Workshop on Formal Methods in Security* (pp. 33-44). ACM Press.

Smith, S., and Thober, M. (2007). Improving usability of information flow security in java. *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security* (pp. 11-20). ACM Press, New York, NY.

Steffan, W., and Clow, J. (1996). Trusted process classes. *Proceedings of the 19th National Information Systems Security Conference*.

Thomas, R., and Sandha, R. (1996). A trusted subject architecture for multilevel secure object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), 16-31.

Thuermer, K. (2007). Managing across domains. *Military Information Technology Online Edition*, 11(6), August 2, 2007.

Volpano, D., and Smith, G. (1997). A type-based approach to program security. *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development* (TAPSOFT'97), 607–621. Springer

Volpano, D., and Smith, G. (1999). Probabilistic noninterference in a concurrent language. *Journal of Computer Security* 7(2,3), 231–253.

Volpano, D., Smith, G., and Irvine, C. (1996). A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3), 167-187.

von Oheimb, D. (2004). Information flow control revisited: Noninfluence = noninterference + nonleakage. *Proceedings of the 9th European Symposium on Research Computer Security*(pp. 225-243). Sophia Antipolis, France.

Wilson, J. (1989). A security policy for an A1 DBMS (a trusted subject).*Proceedings of the 1989 IEEE Symposium on Security and Privacy*, 116-125.

The World Wide Web Consortium (W3C), http://www.w3c.org/. Accessed October 2008.

Zheng, L., and Myers, A. (2004). Dynamic Security Labels and Noninterference. Technical Report 2004-1924, Cornell University.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A – DM-COMPILER RIGAL FILE

This appendix is provided to show the RIGAL compiler files used in defining the DM-Compiler. Because of RIGAL file length limitations, the DM-Compiler is defined using two files: parser.rig and generate.rig. Generally, the first file (*parser.rig*) parses a source base program (*filename.b* file) and generates appropriate Alloy signatures for its program statements. The first file then calls the second RIGAL file (*generate.rig*), which generates the transition predicate for the source base program, defining the semantics of its execution.

RIGAL FILE – PARSER.RIG

```
-------------------------------------------------------------
#main

-- Globals:

--  $variable_table: <* $Id: T *>
--  $constant_list: (. (* Number *) .)
--  $constant_ctr: Number
--  $stmt_num: Number
--  $while_ctr: Number
--  $program: (. (* stmt *) .)
--  $stmt_table: <* SNNN: atomic_stmt *>
-------------------------------------------------------------

  $Parm:= #PARM(T);
  $input_file  := #IMPLODE( $Parm [1] '.b');

  OPEN MSG ' '; -- for error messages

  $output_file := #IMPLODE( $Parm [1] '.txt');
  OPEN GEN $output_file; -- generated Alloy part

  --call the C lexer
  $Lex:= #CALL_PAS( 35 $input_file 'L+A-U-P-C+p-m+');

  MSG<< 'Base Model Generator v.1.0 input from'  $input_file;
  MSG<< '    ' Total #LEN($Lex) tokens;

  $stmt_num := 0;
  $while_ctr := 0;
  $constant_list := (. 0 .); -- 0 always included

  -- do the parsing
  $program:= #ast( $Lex);

  IF $program ->
    MSG<< 'Parsing completed';

    -- add next stmt references
    #add_next_stmt_ref($program);
```

```
    #add_enclosing_cond_ref($program);

    -- start generation
    MSG<< 'Start generation';

    GEN<< '/*********************************/';
    GEN<< '/** DM Implementation Model **/';
    GEN<< '/*********************************/';

    -- generate the source code comment
    #show_program($program);

    -- generate statement signatures
    #gen_stmt_sigs($stmt_table);

    -- generate variable sigs
    #gen_variable_sigs($variable_table);

    -- generate constant sigs
    #gen_constant_sigs($constant_list);

    -- generate state transition predicate
    #gen_state_transition($stmt_table);

    -- generate run commands
    GEN<< '---------------------------';
    GEN<< 'run show for' #LEN($stmt_table) + $while_ctr + 1 'but'
#LEN($variable_table)*($constant_ctr) 'FTuple';
    GEN<< 'check verify_security for' #LEN($stmt_table) + $while_ctr +
1 'but' #LEN($variable_table)*($constant_ctr) 'FTuple';
    GEN<< 'check verify_flow_policy for' #LEN($stmt_table) +
$while_ctr + 1 'but' #LEN($variable_table)*($constant_ctr) 'FTuple';
    GEN<< 'check verify_no_dependency_flaw for' #LEN($stmt_table) +
$while_ctr + 1 'but' #LEN($variable_table)*($constant_ctr) 'FTuple';
    GEN<< 'check verify_no_storage_channel for' #LEN($stmt_table) +
$while_ctr + 1 'but' #LEN($variable_table)*($constant_ctr) 'FTuple';
    GEN<< 'check verify_no_timing_channel for' #LEN($stmt_table) +
$while_ctr + 1 'but' #LEN($variable_table)*($constant_ctr) 'FTuple';

    MSG<< '   ' Total #LEN($stmt_table) statements;

  ELSIF T -> MSG<< 'Errors detected...'
  FI;

##
-----------------------------
#ast
    (. (* $stmt_list !.:= #stmt [';'] *) .)
    /RETURN $stmt_list /
##

#stmt
    Read_dev '(' $al ',' $Id ')'
    /LAST #main $stmt_num +:= 1;
     LAST #main $variable_table ++:= <. $Id: T .>;
     RETURN <.  type:       Read_dev,
          stmt_num: COPY(LAST #main $stmt_num),
          destination: <. var: $Id .>,
          subject_label: $al
         .>/;;

    Write_dev '(' $al ',' $e:= #expr5 ')'
```

```
/LAST #main $stmt_num +:= 1;
 RETURN <.  type: Write_dev,
      stmt_num: COPY(LAST #main $stmt_num),
      source: $e,
      subject_label: $al
     .>/;;

GetDirectFile '(' $al ',' $e:= #expr5 ',' $Id2 ')'
/LAST #main $stmt_num +:= 1;
 LAST #main $variable_table ++:= <. $Id2: T .>;
 RETURN <.  type: GetDirectFile,
      stmt_num: COPY(LAST #main $stmt_num),
      key:        $e,
      destination: <. var: $Id2 .>,
      subject_label: $al
     .>/;;

PutDirectFile '(' $al ',' $e1:= #expr5 ',' $e2:= #expr5 ')'
/LAST #main $stmt_num +:= 1;
 RETURN <.  type: PutDirectFile,
      stmt_num: COPY(LAST #main $stmt_num),
      key:        $e1,
      source:     $e2,
      subject_label: $al
     .>/;;

'{' (* $stmt_list !.:= #stmt * ';' ) [';'] '}'
/RETURN <.  type:       block,
      stmt_list: $stmt_list,
      stmt_num:  $stmt_list[1].stmt_num
     .>/;;

'if' /LAST #main $stmt_num +:= 1;
      $stmt_num:= COPY(LAST #main $stmt_num)/
      $expr:= #expression 'then' $b1:= #stmt [';']
      [ 'else' $b2:= #stmt [';'] ]
/RETURN <.  type: if,
      cond: $expr,
      then_branch: $b1,
      else_branch: $b2,
      stmt_num: $stmt_num,
      var_set: $expr.var_set
      .>/;;

'while' /LAST #main $stmt_num +:= 1;
      LAST #main $while_ctr +:= 1;
      $stmt_num:= COPY(LAST #main $stmt_num)/
      $expr:= #expression 'do' $b1:= #stmt
/RETURN <.  type: while,
      cond: $expr,
      body: $b1,
      stmt_num: $stmt_num,
      var_set: $expr.var_set
      .>/;;

$Id1 ':' '=' $rhp:= #expr5
/LAST #main $stmt_num +:= 1;
 LAST #main $variable_table ++:= <. $Id1: T .>;
 RETURN <.  type:       Assign,
       destination:     <. var: $Id1 .>,
       source:    $rhp,
       stmt_num:  COPY(LAST #main $stmt_num) .>/;;
```

```
    Assign $Id 'from' $e1:= #expr5 'as' $e2:= #expr5
    /LAST #main $stmt_num +:= 1;
     LAST #main $variable_table ++:= <. $Id: T .>;
     RETURN <.  type: Assign,
          stmt_num: COPY(LAST #main $stmt_num),
          destination: <. var: $Id .>,
          source:           $e1,
          source_label:     $e2
         .>/;;

    Stop
    /LAST #main $stmt_num +:= 1;
     RETURN <.  type: Stop,
          stmt_num: COPY(LAST #main $stmt_num)
         .>/;;

    GetClock '(' $al ',' $Id ')'
    /LAST #main $stmt_num +:= 1;
     LAST #main $variable_table ++:= <. $Id: T .>;
     RETURN <.  type:       GetClock,
          stmt_num: COPY(LAST #main $stmt_num),
          destination: <. var: $Id .>,
          subject_label: $al
         .>/;;

    V'($$<>'}') (* $x!.:= S'($$<>';') *)
    /MSG<< Syntax error in statement $x; FAIL/
##

#expression
-- an expression contains a set of variables
    $a1:= #expr2
    (* 'or'  $a2:= #expr2
       /$a1:= <.      arg1: $a1,
          arg2: $a2,
          op: 'or',
          var_set: $a1.var_set ++ $a2.var_set
            .>/ *)
    /RETURN $a1/
##

#expr2
    $a1:= #expr3
    (* 'and'  $a2:= #expr3
       /$a1:= <.      arg1: $a1,
          arg2: $a2,
          op: 'and',
          var_set: $a1.var_set ++ $a2.var_set
            .>/ *)
    /RETURN $a1/
##

#expr3
    'not' $a1:= #expr3
    /RETURN <.  arg: $a1,
          op: 'not',
          var_set: $a1.var_set
         .>/;;

    '(' $a1:= #expression ')'
    /RETURN $a1/;;
```

```
    $e:= #expr4
    /RETURN $e/

##

#expr4
    $a1:= #expr5
       ( ( '>' '=' /$op:= '>='/ ) !
         ( '<' '=' /$op:= '<='/ ) !
        $op:= ( '>' ! '<' ! '=') )
    $a2:= #expr5
    /RETURN <.  arg1: $a1,
          arg2: $a2,
          op:   $op,
          var_set: $a1.var_set ++ $a2.var_set
          .>/;;

    $Id1 $op:= (Before ! LongBefore) $Id2
    /RETURN <. arg1: <. var: $Id1 .>,
          arg2: <. var: $Id2 .>,
          op: $op,
          var_set: <. $Id1: T, $Id2: T .>
          .>/
##

#expr5

    $a1 := ('full' ! 'success')
    /RETURN <. flag: $a1 .>/;;

    $a1 := ('True' ! 'False')
    /RETURN <. bool: $a1 .>/;;

    $a1 := ('SysHigh' ! 'SysMid' ! 'SysLow')
    /RETURN <. src_label: $a1 .>/;;

    $Id
    /LAST #main $variable_table ++:= <. $Id: T .>;
     RETURN  <. var: $Id,
          var_set: <. $Id: T .> .>/;;

    '0'
    /RETURN <. constant: (1-1) .>/;;

    $Num
    /IF NOT #member($Num LAST #main $constant_list) ->
       LAST #main $constant_list !.:= $Num
     FI;
     RETURN <. constant: $Num .>/
##

#member
    $x
    (. (* $y /IF $x = $y -> RETURN T FI/ *) .)
##
-------------------------------------------------
#show_program

    /$indent:= 0;
     GEN<< '-- The base program is below. Total of'
       LAST #main $stmt_num statements/
    (. (* #show_stmt *) .)
```

```
##

#show_stmt

    <. type:    block /GEN<] '{'/,
       stmt_list: (. (* #show_stmt *) .) /GEN<] '}'/
    .>;;


    <. stmt_num: $stmt_num .>
    /GEN<< @ '-- (s' $stmt_num ')  ' #CHR(9);
     #tabs_indent(LAST #show_program $indent);
     FAIL/;;


    <. type: $t:= ( Read_dev ! Write_dev ! GetClock ),
       [ destination: <. var: $Id .> ],
       [ source:      ( <. var: $Id .> ! <. constant: $Num .>) ],
       subject_label: $al
    .>
    /IF $Id ->
       GEN<] $t '(' $al ', ' $Id ');'
     ELSIF T ->
       GEN<] $t '(' $al ', ' $Num ');'
     FI/;;


    <. type: $t:= ( GetDirectFile ! PutDirectFile ),
       key:           ( <. var: $Id .> ! <. constant: $Num .>),
       [ destination: <. var: $Id2 .> ],
       [ source:      ( <. var: $Id2 .> ! <. constant: $Num2 .>) ],
       subject_label: $al
    .>
    /IF $Id ->
       GEN<] $t '(' $al ', ' $Id ','
     ELSIF T ->
       GEN<] $t '(' $al ', ' $Num ','
     FI;
     IF $Id2 ->
       GEN<] $Id2 ');'
     ELSIF T ->
       GEN<] $Num2 ');'
     FI/;;


    <. type: if /GEN<] 'if';
            LAST #show_program $indent +:= 1/,
       cond: #show_expr /GEN<] ' then'/,
       then_branch: #show_stmt,
       [ else_branch:
       / GEN<< @ '-- ' #CHR(9) #CHR(9) ;
         #tabs_indent(LAST #show_program $indent);
         GEN<]'else'/
       #show_stmt ]
    .>
    /LAST #show_program $indent +:= -1/;;


    <. type: while  /GEN<] 'while';
           LAST #show_program $indent +:= 1/,
       cond: #show_expr /GEN<] 'do'/,
       body: #show_stmt
     .>
```

```
    /LAST #show_program $indent +:= -1/;;


    <. type: $t := Assign,
       destination: <. var: $Id .>,
       source: (<. var: $Id2 .> ! <. constant: $Num .>),
       [ source_label: (<. var: $Id3 .> ! <. src_label: $sl .>)]
    .>
    /IF ($Id3 OR $sl) ->
       GEN<] $t $Id 'from';
       IF $Id2 ->
          GEN<] $Id2 'as'
       ELSIF T ->
          GEN<] $Num 'as'
       FI;
       IF $Id3 ->
          GEN<] $Id3 ';'
       ELSIF T ->
          GEN<] $sl ';'
       FI;
    ELSIF T ->
       GEN<] $Id ':=';
          IF $Id2 ->
             GEN<] $Id2 ';'
          ELSIF T ->
             GEN<] $Num ';'
          FI;
    FI/;;


    <. type: Stop  .>
    /GEN<] 'Stop;'/;;


    $x /GEN<< '****  cannot show stmt' $x/

##

#show_expr

    <. flag: $f .>/GEN<] $f/;;

    <. bool: $b .>/GEN<] $b/;;

    <. src_label: $sl .>/GEN<] $sl/;;

    <. var: $Id .> /GEN<] $Id; RETURN T/;;

    <. constant: $Num .> /GEN<] $Num; RETURN T/;;

    <. arg1: /GEN<] '('/ #show_expr,
       op:   $op /GEN<] $op/,
       arg2: #show_expr /GEN<] ')'/
    .>;;

    <. op: 'not' /GEN<] 'not ('/,
       arg: #show_expr  /GEN<] ')'/
    .>;;

    $a /GEN<< cannot show expression $a/

##
```

```
#tabs_indent
    $n
    /$x:= COPY($n);
     LOOP
       IF $x = 0 -> BREAK FI;
       GEN<] @ #CHR(9);
       $x +:= -1
     END/
##
-------------------------------------------------
-- add next stmt reference and create stmt Table
-------------------------------------------------
#add_next_stmt_ref
    $program
    /$n:= 1;
     FORALL $s IN $program DO
       $n +:=1;
       IF $n <= #LEN($program) ->
          $next_stmt:= $program[$n].stmt_num
       FI;
       #add_nextref_to_stmt( $next_stmt $s)
    OD/
##

#add_nextref_to_stmt

    $next

    ( $stmt:= <. type: ( Read_dev ! Write_dev ! GetDirectFile !
PutDirectFile !
               Assign ! GetClock ) .>
       /$stmt ++:= <. next: COPY($next) .>;
        LAST #main $stmt_table ++:= <. #IMPLODE( S $stmt.stmt_num):
$stmt .>/
       !

       $stmt:= <. type: Stop,
             stmt_num: $stmt_num .>
        /$stmt ++:= <. next: $stmt_num .>;
         LAST #main $stmt_table ++:= <. #IMPLODE( S $stmt.stmt_num):
$stmt .>/
       !

       $stmt:= <. type: if,
             then_branch: $then_branch,
             [ else_branch: $else_branch ]
       .>
       /
        #add_nextref_to_stmt( COPY($next) $then_branch);
        IF $else_branch ->
        #add_nextref_to_stmt( COPY($next) $else_branch);
        $stmt ++:= <. next_if_false: $else_branch.stmt_num .>
        ELSIF T ->
        $stmt ++:= <. next_if_false: COPY($next) .>
        FI;
        $stmt ++:= <. next_if_true: $then_branch.stmt_num .>;
        LAST #main $stmt_table ++:= <. #IMPLODE( S $stmt.stmt_num):
$stmt .>/

       !

       $stmt:= <. type: while,
             body: $body
```

```
         .>
       /$stmt ++:= <. next_if_false: COPY($next),
            next_if_true:  $body.stmt_num   .>;
        #add_nextref_to_stmt( COPY($next) $body);
        LAST #main $stmt_table ++:= <. #IMPLODE( S $stmt.stmt_num):
$stmt .>/

       !
        $stmt:= <.     type:       block,
           stmt_list: $list
           .>
        /$stmt ++:= <. next: COPY($next) .>;
         $n:= 1;
         FORALL $s IN $list DO
        $n +:=1;
        IF $n <= #LEN($list) ->
           $next_stmt:= ($list[$n]).stmt_num
        FI;
        #add_nextref_to_stmt( $next_stmt $s)
         OD;
         #add_nextref_to_stmt( $next $list[-1])/
    )

##
-------------------------------------------

-------------------------------------------------
-- add reference to the enclosing if and while stmt
-- to each statement
-------------------------------------------------
#add_enclosing_cond_ref
    $program
    /FORALL $s IN $program DO
       #add_enclosing_refs_to_stmt( NULL $s )
     OD/
##

#add_enclosing_refs_to_stmt

    $enclosing_cond_list

    ( $stmt:= <. type:      if,
          stmt_num:  $stmt_num,
          then_branch:   $then_branch,
          [ else_branch: $else_branch ]
          .>
      /#add_enclosing_refs_to_stmt( (. $stmt_num
.)!!$enclosing_cond_list  $then_branch);
       IF $else_branch ->
       #add_enclosing_refs_to_stmt( (. $stmt_num
.)!!$enclosing_cond_list  $else_branch);
       FI;
       $stmt ++:= <. within_scope_of_cond: $enclosing_cond_list .>/

      !

      $stmt:= <. type: while,
                 body: $body,
                 stmt_num: $while_stmt_num
           .>
       /$stmt ++:= <. next_if_false: COPY($next),
                      next_if_true:  $body.stmt_num   .>;
        #add_nextref_to_stmt( COPY($while_stmt_num) $body);
```

```
        LAST #main $stmt_table ++:= <. #IMPLODE( S $stmt.stmt_num):
$stmt .>/

      !

       $stmt:= <.     type:       block,
          stmt_list:  $list
            .>
       /FORALL $s IN $list DO
       #add_enclosing_refs_to_stmt( $enclosing_cond_list  $s )
        OD /

      !

       $stmt
       /$stmt ++:= <. within_scope_of_cond: $enclosing_cond_list .> /
    )
##
----------------------------------------

#gen_stmt_sigs
    /GEN<<;
     GEN<< '-------------------------';
     GEN<< '/*** Statement sigs ***/';
     GEN<< '-------------------------';/
    <* $s: /GEN<< one sig $s extends Statement '{}';
       GEN<< '{'/
       #gen_stmt_sig
       /GEN<<'}';
       GEN<<;/
    *>
##

#gen_stmt_sig
    <. type: ( (if ! while) /GEN<< #CHR(9) 'type = Condition'/ !
           $t          /GEN<< #CHR(9) 'type =' $t/ ),
       [ source:
       /GEN<< #CHR(9) 'source ='/ $source:= #atomic_expr],
       [ destination:
       /GEN<< #CHR(9) 'destination ='/ $destination:= #atomic_expr],
       [ source_label:
       /GEN<< #CHR(9) 'source_label ='/ $source_label:= #atomic_expr],
       [ key:
       /GEN<< #CHR(9) 'key ='/ $key:= #atomic_expr],
       [ var_set: /GEN<< #CHR(9) 'source ='; $plus:= ' '/
            <* $var: T /GEN<] $plus $var; $plus:= '+'/ *> ],
       [ subject_label: $al ]
    .>
    /IF NOT ($source OR $var) -> GEN<< #CHR(9) 'source = none' FI;
     IF NOT $destination -> GEN<< #CHR(9) 'destination = none' FI;
     IF NOT $source_label -> GEN<< #CHR(9) 'source_label = none' FI;
     IF NOT $key -> GEN<< #CHR(9) 'key = none' FI;
     IF $t = Read_dev OR $t = Write_dev OR $t = GetDirectFile OR $t =
PutDirectFile
            OR $t = GetClock -> GEN<< #CHR(9) 'subject_label = ' $al
FI;
    /
##

#atomic_expr
    <. var: $Id .>/GEN<] $Id; RETURN T/;;
    <. constant: $Num .> /GEN<] @ const $Num; RETURN T/;;
    <. src_label: $sl .>/GEN<] $sl; RETURN T/;;
```

```
##
-------------------------------------------------------
#gen_variable_sigs
     /GEN<< '---------------------------------';
     GEN<< '/*** Variables & Constants ***/';
     GEN<< '---------------------------------';
     GEN<< enum Variable { ;
     GEN<< #CHR(9);
     $comma:= NULL/

    <* $Id: $a /GEN<] $comma @ $Id; $comma:= ','/ *>

    /GEN<< ' }'/
##
-------------------------------------------------------
#gen_constant_sigs
    $const_list
    /$sorted:= #sort($const_list);
     $extended:= #extend($sorted);
     LAST #main $constant_ctr:= #LEN($extended);
     #gen_const($extended);
     #gen_LT_sig($extended)/
##

#sort
    $x
    /$i:= 1;
     LOOP  --just Bubble Sort
       IF $i > (#LEN($x) - 1)  -> BREAK FI;
       $j:= 1;
       LOOP
         IF $j > (#LEN($x) -  $i)  ->   BREAK FI;
         IF $x[$j +1] < $x[$j] ->
       -- swap
       $t:= $x[$j +1]; $x[$j +1]:= $x[$j]; $x[$j]:= $t;
         FI;
         $j +:= 1
       END;
       $i +:=1
     END;
     RETURN $x/
##

#extend
-- extends list $x with number_of_vars constants
    $x
    /$var_count:= #LEN(LAST #main $variable_table);
     $res:= #const_interval($x[1]-$var_count-1  $var_count-1);
     $i:=1;
     LOOP
       IF $i >= #LEN($x) -> BREAK FI;
       IF ($x[$i+1] - $x[$i] - 1) > $var_count ->
          $res !!:= #const_interval($x[$i] $var_count)
       ELSIF  T  ->
          $res !!:= #const_interval($x[$i] ($x[$i+1] - $x[$i] - 1))
       FI;
       $i +:=1
     END;
     $res !!:= #const_interval($x[-1]  $var_count);
     RETURN $res/
##

#const_interval
```

```
-- returns list of $len+1  integers starting with $from
    $from   $len
    /$i:= $from;
     LOOP
       IF $i >= ($from + $len +1) -> BREAK FI;
       $res !.:= COPY($i);
       $i := $i + 1
     END;
     RETURN $res/
##

#gen_const
    /GEN<<;
     GEN<< enum Value { ;
     GEN<< #CHR(9);
     $comma:= NULL;
     $c:=0/
    (. (*    $e
       /IF $e >= 0 ->
          GEN<] @ $comma ' const' $e
        ELSIF  T  ->
          GEN<] @ $comma ' const_minus_' (-$e)
        FI;
        $c +:=1; IF $c MOD 4 = 0 -> GEN<< #CHR(9) FI;
        $comma:= ','/
        *) .)
    /GEN<< ' }'/
##

#gen_LT_sig
    $x
    /GEN<<;
     GEN<< one sig LT '{';
     GEN<] #CHR(9) 'lt:  Value -> Value }';
     GEN<< '{  lt = ^(';
     $plus:= ' ';
     $i:=1;
     LOOP
       IF $i > #LEN($x) -1  -> BREAK FI;
       GEN<< #CHR(9) $plus '(';
       $e:= $x[$i];
       IF $e >= 0 ->
          GEN<] @ ' const' $e
       ELSIF  T  ->
          GEN<] @ ' const_minus_' (-$e)
       FI;
       GEN<] #CHR(9) ' -> ';
       $e:= $x[$i+1];
       IF $e >= 0 ->
          GEN<] @ ' const' $e
       ELSIF  T  ->
          GEN<] @ ' const_minus_' (-$e)
       FI;
       GEN<] ')';
       $plus := '+'; $i +:=1
     END;
     GEN<< ') }'/
##
-----------------------------------------------------------
%INCLUDE <path>/generate.rig
```

RIGAL FILE – GENERATE.RIG

```
-------------------------------------------------------------
-- Generate state transition predicate
-------------------------------------------------------------
#gen_state_transition

    /GEN<<;
     GEN<< '-----------------------------------';
     GEN<< '/*** State Transition Predicate ***/';
     GEN<< '-----------------------------------';
     GEN<< fact trans '{';
     GEN<< #CHR(9) 'all st1: State - InitialState | some st: State |';
     $else:= NULL;
     /
    <* $stmt: /GEN<< #CHR(9) $separator;
          $separator:= ') or';
          GEN<<;
          GEN<< #CHR(9) '( st.stmt =' $stmt '&&';
          GEN<< #CHR(9) '  st1.prev_state = st &&'/
          #gen_stmt_clause
    *>
    /GEN<< #CHR(9) ')';
    GEN<< '}'/
##

#gen_stmt_clause

    <. type:    $t:= Read_dev,
       destination: <. var: $Id .>,
       subject_label: $al,
       next:    $next,
      [ within_scope_of_cond: $enclosing_cond_list]
    .>
    /GEN<< #CHR(9)#CHR(9) '-- ' $t;
     GEN<< #CHR(9)#CHR(9) '( st1.access_label = st.access_label ++ ('
$Id '-> ' $al ' ) &&';
     GEN<< #CHR(9)#CHR(9) 'some n: Value | st1.vars = st.vars ++ ('
       $Id '-> n) &&';
     GEN<< #CHR(9)#CHR(9) @ 'st1.stmt = s' $next ' &&';
     GEN<< #CHR(9)#CHR(9) 'st1.direct_file = st.direct_file &&';
     GEN<< #CHR(9)#CHR(9) 'st1.current_clock =
TO/next[st.current_clock] &&';
     GEN<< #CHR(9)#CHR(9) 'st1.last_cond_checked =
st.last_cond_checked &&';
     GEN<< #CHR(9)#CHR(9) 'st1.influenced_by =';
     GEN<<;
     GEN<< #CHR(9)#CHR(9)#CHR(9) '-- Part A, copy all dependencies for
vars different from' $Id;
     GEN<< #CHR(9)#CHR(9)#CHR(9) '{v: Variable, s: State | (v -> s) in
st.influenced_by && v!=' $Id '}';

     IF $enclosing_cond_list ->
       GEN<<;
       GEN<< #CHR(9)#CHR(9)#CHR(9) '-- Part B, all states from
last_cond_checked';
       GEN<< #CHR(9)#CHR(9)#CHR(9) '-- within which scope this
assignment belongs';
       GEN<< #CHR(9)#CHR(9)#CHR(9) '+ (' $Id '-> {x:
st.last_cond_checked | x.stmt in';
       #print_enclosing_conditions($enclosing_cond_list);
       GEN<] '} )';
```

```
       GEN<<;
       GEN<< #CHR(9)#CHR(9)#CHR(9) '-- Part C, copy dependencies for
all variables participating in';
       GEN<< #CHR(9)#CHR(9)#CHR(9) '-- conditions within which scope
this assignment belongs';
       GEN<< #CHR(9)#CHR(9)#CHR(9) '+ (' $Id '-> State.{ x:
st.last_cond_checked,';
       GEN<< #CHR(9)#CHR(9)#CHR(9)#CHR(9) 'y:
x.influenced_by[x.stmt.source] | x.stmt in';
    #print_enclosing_conditions($enclosing_cond_list);
       GEN<] #CHR(9)#CHR(9)#CHR(9) '} )';
     FI;
     GEN<< #CHR(9)#CHR(9) ')'/;;


    <.type:  $t:= Write_dev,
      source:   ( <.constant: $c .> ! <. var: $Id .> ),
      subject_label: $al,
      next:  $next
    .>
    /GEN<< #CHR(9)#CHR(9) '-- ' $t;
     GEN<< #CHR(9)#CHR(9) '( st1.access_label = st.access_label &&';
     GEN<< #CHR(9)#CHR(9) @ 'st1.stmt = s' $next ' &&';
     GEN<< #CHR(9)#CHR(9) 'st1.direct_file = st.direct_file &&';
     GEN<< #CHR(9)#CHR(9) 'st1.current_clock =
TO/next[st.current_clock] &&';
     GEN<< #CHR(9)#CHR(9) 'st1.influenced_by = st.influenced_by &&';
     GEN<< #CHR(9)#CHR(9) 'st1.last_cond_checked =
st.last_cond_checked';
     GEN<< #CHR(9)#CHR(9) ')'/;;


    <.type:     $t:= Assign,
      destination:    <. var: $Id  .>,
      source:         ( <.var: $Id2 .> ! <.constant: $c .> ),
      [ source_label:       ( <.var: $Id3 .> ! <.src_label: $sl .> )
],
      next:     $next,
      [ within_scope_of_cond: $enclosing_cond_list]
    .>
    /GEN<< #CHR(9)#CHR(9) '-- ';
     IF ($Id3 OR $sl) -> GEN<] 'Trusted';
     ELSIF T -> GEN<] 'Regular'; FI;
     GEN<] $t;
     IF ($Id3 OR $sl) ->
        GEN<< #CHR(9)#CHR(9) '( let xx = tsFilter[ st.vars[' $Id '],
';
        IF $Id2 ->
          GEN<] @ 'st.vars[' $Id2 '], ';
        ELSIF  T  ->
          GEN<] @ 'const' $c ', ';
        FI;
        IF $Id3 ->
          GEN<] @ 'st.vars[' $Id3 '], ';
        ELSIF  T  ->
          GEN<] @ 'const0, ';
        FI;
        GEN<< #CHR(9)#CHR(9)#CHR(9) 'st.access_label[' $Id '], ';
        IF $Id2 ->
          GEN<] 'st.access_label[' $Id2 '], ';
        ELSIF  T  ->
          GEN<] 'SysLow, ';
        FI;
```

```
        IF $Id3 ->
          GEN<] 'st.access_label[' $Id3 ']] | (';
        ELSIF  T  ->
          GEN<] $sl' ] | (';
        FI;
     FI;
     IF ($Id3 OR $sl) ->
        GEN<< #CHR(9)#CHR(9)#CHR(9) 'st1.vars = st.vars ++ (' $Id '->
xx.val ) &&';
        GEN<< #CHR(9)#CHR(9)#CHR(9) 'st1.access_label =
st.access_label ++ (' $Id '-> xx.label )';
        GEN<< #CHR(9)#CHR(9)#CHR(9) ') &&';
     ELSIF $Id2 ->
        GEN<< #CHR(9)#CHR(9) '( st1.vars = st.vars ++ (' $Id '->
st.vars[' $Id2 '] ) &&';
        GEN<< #CHR(9)#CHR(9) 'st1.access_label = st.access_label ++ ('
$Id '-> st.access_label[' $Id2 '] ) &&';
     ELSIF T ->
        GEN<< #CHR(9)#CHR(9) '( st1.vars = st.vars ++ (' $Id @ '->
const' $c ' ) &&';
        GEN<< #CHR(9)#CHR(9) 'st1.access_label = st.access_label ++ ('
$Id '-> SysLow ) &&';
     FI;
     GEN<< #CHR(9)#CHR(9) @ 'st1.stmt = s' $next ' &&';
     GEN<< #CHR(9)#CHR(9) 'st1.direct_file = st.direct_file &&';
     GEN<< #CHR(9)#CHR(9) 'st1.current_clock = st.current_clock &&';
     GEN<< #CHR(9)#CHR(9) 'st1.last_cond_checked =
st.last_cond_checked &&';
     GEN<< #CHR(9)#CHR(9) 'st1.influenced_by =';
     GEN<<;
     GEN<< #CHR(9)#CHR(9)#CHR(9) '-- Part A, copy all dependencies for
vars different from' $Id;
     GEN<< #CHR(9)#CHR(9)#CHR(9) '{v: Variable, s: State | (v -> s) in
st.influenced_by && v!=' $Id '}';
     GEN<<;
     IF $Id3 ->
        GEN<< #CHR(9)#CHR(9)#CHR(9) '-- and inherit all dependencies
of the source_label' $Id3;
        GEN<< #CHR(9)#CHR(9)#CHR(9) '+ (' $Id '-> st.influenced_by['
$Id3 '])';
     ELSIF $Id2 ->
        GEN<< #CHR(9)#CHR(9)#CHR(9) '-- and inherit all dependencies
of the right-hand part' $Id2;
        GEN<< #CHR(9)#CHR(9)#CHR(9) '+ (' $Id '-> st.influenced_by['
$Id2 '])';
     FI;

     IF $enclosing_cond_list ->
       GEN<<;
       GEN<< #CHR(9)#CHR(9)#CHR(9) '-- Part B, all states from
last_cond_checked';
       GEN<< #CHR(9)#CHR(9)#CHR(9) '-- within which scope this
assignment belongs';
       GEN<< #CHR(9)#CHR(9)#CHR(9) '+ (' $Id '-> {x:
st.last_cond_checked | x.stmt in';
       #print_enclosing_conditions($enclosing_cond_list);
       GEN<] '} )';
       GEN<<;
       GEN<< #CHR(9)#CHR(9)#CHR(9) '-- Part C, copy dependencies for
all variables participating in';
       GEN<< #CHR(9)#CHR(9)#CHR(9) '-- conditions within which scope
this assignment belongs';
```

```
       GEN<< #CHR(9)#CHR(9)#CHR(9) '+ (' $Id '-> State.{ x:
st.last_cond_checked,';
       GEN<< #CHR(9)#CHR(9)#CHR(9)#CHR(9) 'y:
x.influenced_by[x.stmt.source] | x.stmt in';
       #print_enclosing_conditions($enclosing_cond_list);
       GEN<] ' } )';
     FI;
     GEN<< #CHR(9)#CHR(9) ')'/;;


    <. type:    $t:= ( if ! while ),
       cond:    $expr,
       next_if_false: $next_if_false,
       next_if_true:  $next_if_true,
       stmt_num:      $stmt_num
    .>
    /GEN<< #CHR(9)#CHR(9) '-- ' $t;
     GEN<< #CHR(9)#CHR(9) '( st1.access_label = st.access_label &&';
     GEN<< #CHR(9)#CHR(9) 'st1.vars = st.vars  &&';
     GEN<< #CHR(9)#CHR(9) 'st1.current_clock = st.current_clock  &&';
     GEN<< #CHR(9)#CHR(9) 'st1.direct_file = st.direct_file  &&';
     GEN<< #CHR(9)#CHR(9) 'st1.influenced_by = st.influenced_by &&';
     GEN<< #CHR(9)#CHR(9) @ 'st1.last_cond_checked = {x:
st.last_cond_checked | x.stmt != s' $stmt_num '} + st &&';
     GEN<< #CHR(9)#CHR(9) '(';
     #gen_expr($expr);
     GEN<< #CHR(9)#CHR(9)#CHR(9) @ ' => st1.stmt = s' $next_if_true;
     GEN<< #CHR(9)#CHR(9)#CHR(9) @ ' else st1.stmt = s' $next_if_false
')';
     GEN<< #CHR(9)#CHR(9) ')'/;;



    <.type:     $t:= GetDirectFile,
     key:    ( <.var: $Id .> ! <.constant: $c .> ),
     destination:     <.var: $Id2 .>,
      subject_label: $al,
     next:   $next
    .>
    /GEN<< #CHR(9)#CHR(9) '-- ' $t;
     GEN<< #CHR(9)#CHR(9) @ '( st1.stmt = s' $next ' &&';
     GEN<< #CHR(9)#CHR(9) '  st1.current_clock =
TO/next[st.current_clock] &&';
     GEN<< #CHR(9)#CHR(9) '  st1.direct_file.keyContent =
st.direct_file.keyContent  &&';
     GEN<< #CHR(9)#CHR(9) '  st1.direct_file.keyLabel =
st.direct_file.keyLabel  &&';
     GEN<< #CHR(9)#CHR(9) '  st1.last_cond_checked =
st.last_cond_checked &&';
     GEN<< #CHR(9)#CHR(9) '  st1.direct_file.full =
st.direct_file.full &&';

     IF $Id ->
       GEN<< #CHR(9)#CHR(9) '  ( (st.vars[' $Id '] in
st.direct_file.keyContent.Value) =>'
     ELSIF  T  ->
       GEN<< #CHR(9)#CHR(9) @ '  ( (const' $c ' in
st.direct_file.keyContent.Value) =>'
     FI;

     GEN<< #CHR(9)#CHR(9) '     -- the key is found';
     GEN<< #CHR(9)#CHR(9) '     ( st1.access_label = st.access_label
++ (' $Id2 '-> ' $al ' ) &&';
```

```
     GEN<< #CHR(9)#CHR(9) '       st1.vars = st.vars ++ ';
     IF $Id ->
       GEN<< #CHR(9)#CHR(9) '             (' $Id2 '->
st.direct_file.keyContent[st.vars[' $Id ']] ) &&';
     ELSIF  T  ->
       GEN<< #CHR(9)#CHR(9) @ '           ( ' $Id2 ' ->
st.direct_file.keyContent[const' $c '] ) &&';
     FI;

     GEN<< #CHR(9)#CHR(9) '       st1.direct_file.success = const1 )';
     GEN<< #CHR(9)#CHR(9) '    else  -- the key is not found';
     GEN<< #CHR(9)#CHR(9) '     ( st1.vars = st.vars &&';
     GEN<< #CHR(9)#CHR(9) '       st1.access_label = st.access_label
&&';
     GEN<< #CHR(9)#CHR(9) '       st1.direct_file.success = const0 )';
     GEN<< #CHR(9)#CHR(9)   '  )';
     GEN<< #CHR(9)#CHR(9)   ')'/;;


    <.type:     $t:= PutDirectFile,
      key:      ( <.var: $Id  .> ! <.constant: $c  .> ),
     source: ( <.var: $Id2 .> ! <.constant: $c2 .> ),
      subject_label: $al,
     next:   $next
    .>
    /GEN<< #CHR(9)#CHR(9) '-- ' $t;
     GEN<< #CHR(9)#CHR(9) @ '( st1.stmt = s' $next ' &&';
     GEN<< #CHR(9)#CHR(9) '  st1.current_clock =
TO/next[st.current_clock] &&';
     GEN<< #CHR(9)#CHR(9) '  st1.last_cond_checked =
st.last_cond_checked &&';
     GEN<< #CHR(9)#CHR(9) '  st1.vars = st.vars &&';
     GEN<< #CHR(9)#CHR(9) '  st1.access_label = st.access_label &&';

     IF $Id ->
       GEN<< #CHR(9)#CHR(9) '  ( (st.vars[' $Id '] in
st.direct_file.keyContent.Value) =>'
     ELSIF  T  ->
       GEN<< #CHR(9)#CHR(9) @ '  ( (const' $c ' in
st.direct_file.keyContent.Value) =>'
     FI;

     GEN<< #CHR(9)#CHR(9) '     -- the key is found';
     GEN<< #CHR(9)#CHR(9) '       (st1.direct_file.success = const1
&&';

     GEN<< #CHR(9)#CHR(9) '        st1.direct_file.keyContent =
st.direct_file.keyContent ++';
     IF $Id ->
       GEN<<  #CHR(9)#CHR(9)#CHR(9) '           (st.vars[' $Id '] ->
';
     ELSIF  T  ->
       GEN<<  #CHR(9)#CHR(9)#CHR(9) @ '         ( const' $c ' -> ';
     FI;
     IF $Id2 ->
       GEN<]  'st.vars[' $Id2 ']) &&';
     ELSIF  T  ->
       GEN<]  @ const $c2 ') &&';
     FI;

     GEN<< #CHR(9)#CHR(9) '        st1.direct_file.keyLabel =
st.direct_file.keyLabel ++';
     IF $Id ->
```

```
       GEN<<  #CHR(9)#CHR(9)#CHR(9) '           (st.vars[' $Id '] ->
';
     ELSIF  T  ->
       GEN<<  #CHR(9)#CHR(9)#CHR(9) @ '         ( const' $c ' -> ';
     FI;
     GEN<] $al ' ) &&';
     GEN<< #CHR(9)#CHR(9) '        -- since key already existed, full
remains the same';
     GEN<< #CHR(9)#CHR(9) '        st1.direct_file.full =
st.direct_file.full';
     GEN<< #CHR(9)#CHR(9) '       )';
     GEN<< #CHR(9)#CHR(9) '      else -- the key is not found';
     GEN<< #CHR(9)#CHR(9) '       ( st.direct_file.full = const0 =>
-- Direct File not Full';

     GEN<< #CHR(9)#CHR(9) '         ( st1.direct_file.keyContent =
st.direct_file.keyContent ++';
     IF $Id ->
       GEN<<  #CHR(9)#CHR(9)#CHR(9) '           (st.vars[' $Id '] ->
';
     ELSIF  T  ->
       GEN<<  #CHR(9)#CHR(9)#CHR(9) @ '         ( const' $c ' -> ';
     FI;
     IF $Id2 ->
       GEN<]  'st.vars[' $Id2 ']) &&';
     ELSIF  T  ->
       GEN<]  @ const $c2 ') &&';
     FI;

     GEN<< #CHR(9)#CHR(9) '          st1.direct_file.keyLabel =
st.direct_file.keyLabel ++';
     IF $Id ->
       GEN<<  #CHR(9)#CHR(9)#CHR(9) '           (st.vars[' $Id '] ->
';
     ELSIF  T  ->
       GEN<<  #CHR(9)#CHR(9)#CHR(9) @ '         ( const' $c ' -> ';
     FI;
     GEN<] $al ' ) &&';
     GEN<< #CHR(9)#CHR(9) '          st1.direct_file.success = const1
&&';
     GEN<< #CHR(9)#CHR(9) '          -- if content limit reached, set
full to const1 (true)';
     GEN<< #CHR(9)#CHR(9) '          (#st1.direct_file.keyContent =
st1.direct_file.max_slots => ';
     GEN<< #CHR(9)#CHR(9) '             st1.direct_file.full = const1
else st1.direct_file.full = const0)';
     GEN<< #CHR(9)#CHR(9) '         )';
     GEN<< #CHR(9)#CHR(9) '         else  -- Direct File is Full';
     GEN<< #CHR(9)#CHR(9) '          (st1.direct_file = st.direct_file
&&';
     GEN<< #CHR(9)#CHR(9) '           st1.direct_file.success = const0
&&';
     GEN<< #CHR(9)#CHR(9) '           -- assign full to const1
(true)';
     GEN<< #CHR(9)#CHR(9) '           st1.direct_file.full = const1)';
     GEN<< #CHR(9)#CHR(9) '       )';
     GEN<< #CHR(9)#CHR(9) '   )';
     GEN<< #CHR(9)#CHR(9) ' )'/;;


    <.type:     $t:= GetClock,
      destination:    <.var: $Id .>,
      subject_label: $al,
```

```
      next:     $next .>
    /GEN<< #CHR(9)#CHR(9) '-- ' $t;
     GEN<< #CHR(9)#CHR(9) '( st1.access_label = st.access_label ++ ('
$Id '-> ' $al ' ) &&';
     GEN<< #CHR(9)#CHR(9) 'st1.vars = st.vars ++ (' $Id @ '->
st.current_clock ) &&';
     GEN<< #CHR(9)#CHR(9) @ 'st1.stmt = s' $next ' &&';
     GEN<< #CHR(9)#CHR(9) 'st1.direct_file = st.direct_file &&';
     GEN<< #CHR(9)#CHR(9) 'st1.current_clock = st.current_clock &&';
     GEN<< #CHR(9)#CHR(9) 'st1.last_cond_checked =
st.last_cond_checked';
     GEN<< #CHR(9)#CHR(9) ')'/;;


    <. type: $t:= Stop .>
    /GEN<< #CHR(9)#CHR(9) '-- ' $t;
     GEN<< #CHR(9)#CHR(9) '( st1.stmt = st.stmt )'/;;

    <. type: $t .> /GEN<< #CHR(9)#CHR(9) '******' stmt type $t not yet
implemented '******'/

##

#print_enclosing_conditions
    /$plus:= ' '/
    (. (* $c /GEN<] @ $plus 'S' $c; $plus:= '+'/ *) .)
##
-----------------------------------------------------------
#gen_expr

    <. flag: $f .>/GEN<] @ 'st.direct_file.'$f' '/;;

    <. bool: $b .>
    /IF $b = True ->
       GEN<] 'const1';
     ELSIF $b = False ->
       GEN<] 'const0';
     FI/;;

    <. var: $Id .>/GEN<] 'st.vars[' $Id ']'/;;

    <. constant: $Num .> /GEN<] @ const $Num/;;

    <. op:    $op:= ( or ! and ! '=' ),
       arg1:  /GEN<] '('/ #gen_expr /GEN<] $op/,
       arg2:  #gen_expr /GEN<] ')'/
    .> ;;

    <. op:    '<',
       arg1: /GEN<< #CHR(9) #CHR(9) #CHR(9) '(('/ #gen_expr,
       arg2: /GEN<] '->'/ #gen_expr
    .>
    /GEN<] ') in LT.lt)'/;;

    <. op:    '>',
       arg2: /GEN<< #CHR(9) #CHR(9) #CHR(9)  '(('/ #gen_expr,
       arg1: /GEN<] '->'/ #gen_expr
    .>
    /GEN<] ') in LT.lt)'/;;

    <. op:    '<=',
       arg1: /GEN<< #CHR(9) #CHR(9) #CHR(9) '(('/ #gen_expr,
       arg2: /GEN<] '->'/ #gen_expr /GEN<] ') in LT.lt or'/,
```

```
      arg1:  #gen_expr,
      arg2: /GEN<] '='/ #gen_expr /GEN<] ')'/
   .>;;

   <. op:    '>=',
      arg2: /GEN<< #CHR(9) #CHR(9) #CHR(9) '(('/ #gen_expr,
      arg1: /GEN<] '->'/ #gen_expr /GEN<] ') in LT.lt or'/,
      arg1:  #gen_expr,
      arg2: /GEN<] '='/ #gen_expr /GEN<] ')'/
   .>;;

   <. op:    Before,
      arg1: /GEN<< #CHR(9) #CHR(9) #CHR(9) '(('/ #gen_expr,
      arg2: /GEN<] '->'/ #gen_expr
   .>
   /GEN<] ') in Clock.before)'/;;

   <. op:    LongBefore,
      arg1: /GEN<< #CHR(9) #CHR(9) #CHR(9) '(('/ #gen_expr,
      arg2: /GEN<] '->'/ #gen_expr
   .>
   /GEN<] ') in Clock.long_before)'/;;

   <. op: 'not' /GEN<< #CHR(9)  #CHR(9)  #CHR(9)  'not ('/,
      arg: #gen_expr  /GEN<] ')'/
   .>;;

   $e /GEN<< #CHR(9) '***' expression #show_expr($e) not implemented
'***'/

##
```

APPENDIX B.1 – GENERATED DM FOR BASE PROGRAM EXAMPLE 1

This appendix provides complete code for the overt control dependency flaw example base program and resultant DM described in Chapter VI - "Example DM Implementations." The DM below is generated by the DM-Compiler from the following base program:

```
(s1)      Read_dev (SysHigh, x1);
(s2)      if x1 = 0 then
(s3)         x2:= 0;
(s4)      else x2:= 1;
(s5)      Write_dev (SysLow, x2);
(s6)      Stop;
```

The Alloy specification for the DM follows:

```
/*************************************************************************/
module static_model
open util/ordering[Time] as TO
/*************************************************************************/

/*************************/
/** DM Invariant Model **/
/*************************/

sig Statement {
    type:  Stmt_type,
    destination: lone Variable,
    source:  set Variable + Value,
    source_label: lone (AccessLabel + Variable),
    key:   lone (Variable + Value),
    subject_label:  lone AccessLabel
}

enum Stmt_type {
    Assign, Condition,
    Read_dev, Write_dev,
    GetDirectFile, PutDirectFile,
    GetClock, Stop
}

-- define access labels based on security policy lattice
enum AccessLabel { SysHigh, SysMid, SysLow }

-- define a Policy signature to allow BLP-style info flows
one sig Policy {
    ord: AccessLabel -> AccessLabel
}
{   ord = ^( (SysLow -> SysMid)
         + (SysMid -> SysHigh) )
         + (iden & (AccessLabel -> AccessLabel) )
}
```

```
sig State {
    stmt:  Statement, -- next stmt to execute
    vars:  Variable -> one (Value + Time), -- variable table
    access_label:  Variable -> one AccessLabel,
    direct_file:  DirectFile, -- current snapshot
    current_clock: Time,
    prev_state: lone State,
    err_msg:  lone Error,
    influenced_by: Variable -> State,
    last_cond_checked: set State,
}
{ -- define error conditions
    ( err_msg = InfoFlow_error <=>
       not consistent_with_FlowPolicy [this] ) &&
    ( err_msg = Overt_flaw_detected <=>
       dependency_flaw_found[this] ) &&
    ( err_msg = Storage_channel_detected <=>
       storage_channel_found[this] ) &&
    ( err_msg = Timing_channel_detected <=>
       timing_channel_found[this] )
}

-- Signature for error types
enum Error {
    InfoFlow_error,
    Overt_flaw_detected,
    Storage_channel_detected,
    Timing_channel_detected
}

-----------------------------------------------------------------------
-- Initialization of State signature: all variables initially have 0
--  value and SysLow label, and DirectFile is empty
one sig InitialState extends State {}
{
    vars = (Variable -> const0)
    access_label = (Variable -> SysLow)
    stmt = S1
    direct_file.full = const0
    direct_file.success =const1
    current_clock = TO/first[]
    prev_state =  none
    err_msg =  none
    last_cond_checked = none
    no influenced_by
    no direct_file.keyContent
    no direct_file.keyLabel
}

-- Sig establishes ordering of States in a program execution
one sig State_order {
    st_after: State -> State
}
{   st_after = ^ prev_state
}

-- a "Stop" State cannot precede another State
fact { all s: State | s.prev_state.stmt.type != Stop }

-- no two States can be identical
fact { no disj st1, st2: State |
    (st1.stmt = st2.stmt  &&
     st1.prev_state = st2.prev_state &&
```

```
     st1.vars = st2.vars &&
     st1.direct_file = st2.direct_file)
}

-------------------------------------------------------------------
sig DirectFile {
    -- each key Value is assigned a content Value and AccessLabel
    keyContent:Value -> lone Value,
    keyLabel: Value -> lone AccessLabel,
    last_written: lone AccessLabel,
    full:  (const0 + const1),
    success: (const0 + const1),
    max_slots: Int
}
{      max_slots = 2     -- capacity limited to 2 key locations
}

-------------------------------------------------------------------
sig Time {}

one sig Clock {
    before:  Time -> Time,
    long_before: Time -> Time
}
{ long_before in before &&
    all t1: Time, t2: Time - t1 |
    ((t1->t2) in before <=> t2 in TO/nexts[t1]) &&
    ((t1->t2) in long_before <=> some t3: Time |
       (t3 in before[t1] && t3 in before.t2))
}

-------------------------------------------------------------------
-- Alloy signature used for passing results of tsFilter function
sig FTuple {
    val: Value,
    label: AccessLabel
}

fact { all v: Value, a: AccessLabel | one f: FTuple |
    f.val=v && f.label=a }

-------------------------------------------------------------------
-- Functions, Facts, Assertions and Predicates for info flow security
--  policy and security rules
-------------------------------------------------------------------
-- The tsFilter function defines the semantics of the Trusted Subj
--  Assignment statement, by enabling a TS to act as a Content
--  or Label Filter.
-- Different invariant models may define different filter functions,
--  depending on the TS semantics that must be demonstrated.
fun tsFilter[dv, s1v, s2v: Value,
             da, s1a, s2a: AccessLabel]: FTuple {
{ result: FTuple | {
    result.val = (((s1v->const0) in LT.lt)
        => const0 else s1v)
    result.label = (((da->s2a) in Policy.ord)
        => s2a else
      (((s2a->SysMid) in Policy.ord)
        => SysMid else s2a)) }
} }

-------------------------------------------------------------------
-- Security assertion to verify program abides by all security rules
```

```
--  assert verify_security {
    all s: State |
       consistent_with_FlowPolicy [s] &&
       not dependency_flaw_found [s] &&
       not storage_channel_found [s] &&
       not timing_channel_found [s]
}

-----------------------------------------------------------------
-- Define how statements abide by info flow policy
assert verify_flow_policy {
    all s: State | consistent_with_FlowPolicy[s] }

pred consistent_with_FlowPolicy [s: State] {
    let stm = s.stmt | {
       -- for Write_dev or PutDirectFile statement
       (stm.type in (Write_dev + PutDirectFile) &&
         stm.source in Variable)
       => ((s.access_label[stm.source] -> stm.subject_label)
           in Policy.ord)
    }
}

-----------------------------------------------------------------
-- Verify no overt control dependency flaw found in current State

assert verify_no_dependency_flaw {
    all s: State | not dependency_flaw_found[s] }

-- Define conditions under which a control dependency flaw could
--  exist; checks for a Write, where source in the current state is
--  influenced_by State with higher label in required access.
--  Assertion uses dynamic slicing techniques.
pred dependency_flaw_found [s: State] {
    let stm = s.stmt, s1 = s.influenced_by[stm.source] | {
       stm.type = Write_dev  &&
       stm.source in Variable &&
       -- check if Write_dev source was influenced_by a var
       --  higher than subject
       not ((s1.access_label[s1.stmt.source] -> stm.subject_label)
           in Policy.ord)
    }
}

-----------------------------------------------------------------
-- Verify no storage covert channels found in current State

assert verify_no_storage_channel {
    all s: State |
       not storage_channel_found[s] }

-- Define conditions under which a storage channel could exist icw
--  a PutDirectFile
pred storage_channel_found [s: State] {
    let stm = s.stmt | {
       stm.type = PutDirectFile &&
       s.direct_file.full = const1 &&
       -- check if direct file was last written by a higher subject
       not ((s.direct_file.last_written -> stm.subject_label)
         in Policy.ord)
    }
}
```

```
-------------------------------------------------------------------
-- Verify no timing covert channels found in current State

assert verify_no_timing_channel {
    all s: State | not timing_channel_found[s] }

-- Define conditions under which a timing channel could exist
pred timing_channel_found [gc2: State] {
    some disj rw, gc1: State | {
       (gc2 -> rw) in State_order.st_after &&
       (rw -> gc1) in State_order.st_after &&
       gc1.stmt.type = GetClock &&
       gc2.stmt.type = GetClock &&
       rw.stmt.type in (Read_dev + Write_dev
         + PutDirectFile + GetDirectFile) &&
       -- check if GetClocks are at same level
       gc1.stmt.subject_label = gc2.stmt.subject_label &&
       -- check if Read/Write/DirectFile operation at
       --  higher level than GetClock
       not ((rw.stmt.subject_label -> gc2.stmt.subject_label)
         in Policy.ord)
    }
}

-------------------------------------------------------------------
-- Find a consistent instance of this model
pred show () {}
-------------------------------------------------------------------
/*******************************/
/** DM Implementation Model **/
/*******************************/
-- The base program is below. Total of 6 statements
-- (S1)   Read_dev ( SysHigh ,  x1 );
-- (S2)   if ( x1 = 0 )  then
-- (S3)    x2 := 0 ;
--        else
-- (S4)    x2 := 1 ;
-- (S5)   Write_dev ( SysLow ,  x2 );
-- (S6)   Stop;

-------------------------
/*** Statement sigs ***/
-------------------------
one sig S1 extends Statement {}
{
     type = Read_dev
     destination = x1
     source = none
     source_label = none
     key = none
     subject_label =  SysHigh
}

one sig S3 extends Statement {}
{
     type = Assign
     source = const0
     destination = x2
     source_label = none
     key = none
}

one sig S4 extends Statement {}
```

```
{
     type = Assign
     source = const1
     destination = x2
     source_label = none
     key = none
}

one sig S2 extends Statement {}
{
     type = Condition
     source =   x1
     destination = none
     source_label = none
     key = none
}

one sig S5 extends Statement {}
{
     type = Write_dev
     source = x2
     destination = none
     source_label = none
     key = none
     subject_label =  SysLow
}

one sig S6 extends Statement {}
{
     type = Stop
     source = none
     destination = none
     source_label = none
     key = none
}

---------------------------------
/*** Variables & Constants ***/
---------------------------------
enum Variable {
     x1, x2
 }

enum Value {
      const_minus_3, const_minus_2, const0, const1
     , const2, const3
 }

one sig LT {  lt:  Value -> Value }
{  lt = ^(
       (  const_minus_3       ->   const_minus_2)
     + (  const_minus_2       ->   const0)
     + (  const0   ->   const1)
     + (  const1   ->   const2)
     + (  const2   ->   const3)
) }

-----------------------------------
/*** State Transition Predicate ***/
-----------------------------------
fact trans {
     all st1: State - InitialState | some st: State |
```

```
     ( st.stmt = S1 &&
       st1.prev_state = st &&
        --  Read_dev
        ( st1.access_label = st.access_label ++ ( x1 ->  SysHigh  ) &&
        some n: Value | st1.vars = st.vars ++ ( x1 -> n) &&
        st1.stmt = S2 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

          -- Part A, copy all dependencies for vars different from x1
          {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x1 }
        )
     ) or

     ( st.stmt = S3 &&
       st1.prev_state = st &&
        --  Regular Assign
        ( st1.vars = st.vars ++ ( x2 -> const0 ) &&
        st1.access_label = st.access_label ++ ( x2 -> SysLow ) &&
        st1.stmt = S5 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = st.current_clock &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

          -- Part A, copy all dependencies for vars different from x2
          {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x2 }


          -- Part B, all states from last_cond_checked
          -- within which scope this assignment belongs
          + ( x2 -> {x: st.last_cond_checked | x.stmt in  S2} )

          -- Part C, copy dependencies for all variables
participating in
          -- conditions within which scope this assignment belongs
          + ( x2 -> State.{ x: st.last_cond_checked,
            y: x.influenced_by[x.stmt.source] | x.stmt in  S2 } )
        )
     ) or

     ( st.stmt = S4 &&
       st1.prev_state = st &&
        --  Regular Assign
        ( st1.vars = st.vars ++ ( x2 -> const1 ) &&
        st1.access_label = st.access_label ++ ( x2 -> SysLow ) &&
        st1.stmt = S5 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = st.current_clock &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

          -- Part A, copy all dependencies for vars different from x2
          {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x2 }


          -- Part B, all states from last_cond_checked
```

```
          -- within which scope this assignment belongs
          + ( x2 -> {x: st.last_cond_checked | x.stmt in  S2} )

          -- Part C, copy dependencies for all variables
participating in
          -- conditions within which scope this assignment belongs
          + ( x2 -> State.{ x: st.last_cond_checked,
            y: x.influenced_by[x.stmt.source] | x.stmt in  S2 } )
        )
     ) or

     ( st.stmt = S2 &&
       st1.prev_state = st &&
        --  if
        ( st1.access_label = st.access_label &&
        st1.vars = st.vars  &&
        st1.current_clock = st.current_clock  &&
        st1.direct_file = st.direct_file  &&
        st1.influenced_by = st.influenced_by &&
        st1.last_cond_checked = {x: st.last_cond_checked | x.stmt !=
S2} + st &&
        ( ( st.vars[ x1 ] = const0)
           => st1.stmt = S3
           else st1.stmt = S4)
        )
     ) or

     ( st.stmt = S5 &&
       st1.prev_state = st &&
        --  Write_dev
        ( st1.access_label = st.access_label &&
        st1.stmt = S6 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.influenced_by = st.influenced_by &&
        st1.last_cond_checked = st.last_cond_checked
        )
     ) or

     ( st.stmt = S6 &&
       st1.prev_state = st &&
        --  Stop
        ( st1.stmt = st.stmt )
     )
}
--------------------------
run show for 7 but 18 FTuple
check verify_security for 7 but 18 FTuple
check verify_flow_policy for 7 but 18 FTuple
check verify_no_dependency_flaw for 7 but 18 FTuple
check verify_no_storage_channel for 7 but 18 FTuple
check verify_no_timing_channel for 7 but 18 FTuple
```

APPENDIX B.2 – GENERATED DM FOR BASE PROGRAM EXAMPLE 2

This appendix provides complete code for the timing covert channel example base program and resultant DM described in Chapter VI - "Example DM Implementations." The DM below is generated by the DM-Compiler from the following base program:

```
(s1)      Read_dev (SysHigh, x1);
(s2)      GetClock (SysLow, t1);
(s3)      if x1 < 0 then
(s4)         PutDirectFile (SysHigh, 1, x1);
(s5)      GetClock(SysLow, t2);
(s6)      if t1 Before t2 then
(s7)         Write_dev (SysLow, 1);
(s8)      else Write_dev (SysLow, 0);
(s9)      Stop;
```

The Alloy specification for the DM follows:

```
/*****************************************************************/
module static_model
open util/ordering[Time] as TO
/*****************************************************************/

/*************************/
/** DM Invariant Model **/
/*************************/

sig Statement {
    type:  Stmt_type,
    destination: lone Variable,
    source:  set Variable + Value,
    source_label: lone (AccessLabel + Variable),
    key:   lone (Variable + Value),
    subject_label: lone AccessLabel
}

enum Stmt_type {
    Assign, Condition,
    Read_dev, Write_dev,
    GetDirectFile, PutDirectFile,
    GetClock, Stop
}

-- define access labels based on security policy lattice
enum AccessLabel { SysHigh, SysMid, SysLow }

-- define a Policy signature to allow BLP-style info flows
one sig Policy {
    ord: AccessLabel -> AccessLabel
}
{   ord = ^( (SysLow -> SysMid)
         + (SysMid -> SysHigh) )
         + (iden & (AccessLabel -> AccessLabel) )
}
```

```
sig State {
    stmt:  Statement, -- next stmt to execute
    vars:  Variable -> one (Value + Time), -- variable table
    access_label:  Variable -> one AccessLabel,
    direct_file:  DirectFile, -- current snapshot
    current_clock: Time,
    prev_state: lone State,
    err_msg:  lone Error,
    influenced_by: Variable -> State,
    last_cond_checked: set State,
}
{ -- define error conditions
    ( err_msg = InfoFlow_error <=>
       not consistent_with_FlowPolicy [this] ) &&
    ( err_msg = Overt_flaw_detected <=>
       dependency_flaw_found[this] ) &&
    ( err_msg = Storage_channel_detected <=>
       storage_channel_found[this] ) &&
    ( err_msg = Timing_channel_detected <=>
       timing_channel_found[this] )
}

-- Signature for error types
enum Error {
    InfoFlow_error,
    Overt_flaw_detected,
    Storage_channel_detected,
    Timing_channel_detected
}

-----------------------------------------------------------------------
-- Initialization of State signature: all variables initially have 0
--  value and SysLow label, and DirectFile is empty
one sig InitialState extends State {}
{
    vars = (Variable -> const0)
    access_label = (Variable -> SysLow)
    stmt = S1
    direct_file.full = const0
    direct_file.success =const1
    current_clock = TO/first[]
    prev_state =  none
    err_msg =  none
    last_cond_checked = none
    no influenced_by
    no direct_file.keyContent
    no direct_file.keyLabel
}

-- Sig establishes ordering of States in a program execution
one sig State_order {
    st_after: State -> State
}
{   st_after = ^ prev_state
}

-- a "Stop" State cannot precede another State
fact { all s: State | s.prev_state.stmt.type != Stop }

-- no two States can be identical
fact { no disj st1, st2: State |
    (st1.stmt = st2.stmt  &&
```

```
     st1.prev_state = st2.prev_state &&
     st1.vars = st2.vars &&
     st1.direct_file = st2.direct_file)
}

-------------------------------------------------------------------
sig DirectFile {
    -- each key Value is assigned a content Value and AccessLabel
    keyContent:Value -> lone Value,
    keyLabel: Value -> lone AccessLabel,
    last_written: lone AccessLabel,
    full:  (const0 + const1),
    success: (const0 + const1),
    max_slots: Int
}
{      max_slots = 2     -- capacity limited to 2 key locations
}

-------------------------------------------------------------------
sig Time {}

one sig Clock {
    before:  Time -> Time,
    long_before: Time -> Time
}
{ long_before in before &&
    all t1: Time, t2: Time - t1 |
    ((t1->t2) in before <=> t2 in TO/nexts[t1]) &&
    ((t1->t2) in long_before <=> some t3: Time |
       (t3 in before[t1] && t3 in before.t2))
}

-------------------------------------------------------------------
-- Alloy signature used for passing results of tsFilter function
sig FTuple {
    val: Value,
    label: AccessLabel
}

fact { all v: Value, a: AccessLabel | one f: FTuple |
    f.val=v && f.label=a }

-------------------------------------------------------------------
-- Functions, Facts, Assertions and Predicates for info flow security
--  policy and security rules
-------------------------------------------------------------------
-- The tsFilter function defines the semantics of the Trusted Subj
--  Assignment statement, by enabling a TS to act as a Content
--  or Label Filter.
-- Different invariant models may define different filter functions,
--  depending on the TS semantics that must be demonstrated.
fun tsFilter[dv, s1v, s2v: Value,
             da, s1a, s2a: AccessLabel]: FTuple {
{ result: FTuple | {
    result.val = (((s1v->const0) in LT.lt)
        => const0 else s1v)
    result.label = (((da->s2a) in Policy.ord)
        => s2a else
      (((s2a->SysMid) in Policy.ord)
        => SysMid else s2a)) }
} }

-------------------------------------------------------------------
```

```
-- Security assertion to verify program abides by all security rules
--  assert verify_security {
    all s: State |
       consistent_with_FlowPolicy [s] &&
       not dependency_flaw_found [s] &&
       not storage_channel_found [s] &&
       not timing_channel_found [s]
}

-----------------------------------------------------------------
-- Define how statements abide by info flow policy
assert verify_flow_policy {
    all s: State | consistent_with_FlowPolicy[s] }

pred consistent_with_FlowPolicy [s: State] {
    let stm = s.stmt | {
       -- for Write_dev or PutDirectFile statement
       (stm.type in (Write_dev + PutDirectFile) &&
         stm.source in Variable)
       => ((s.access_label[stm.source] -> stm.subject_label)
           in Policy.ord)
    }
}

-----------------------------------------------------------------
-- Verify no overt control dependency flaw found in current State

assert verify_no_dependency_flaw {
    all s: State | not dependency_flaw_found[s] }

-- Define conditions under which a control dependency flaw could
--  exist; checks for a Write, where source in the current state is
--  influenced_by State with higher label in required access.
--  Assertion uses dynamic slicing techniques.
pred dependency_flaw_found [s: State] {
    let stm = s.stmt, s1 = s.influenced_by[stm.source] | {
       stm.type = Write_dev  &&
       stm.source in Variable &&
       -- check if Write_dev source was influenced_by a var
       --  higher than subject
       not ((s1.access_label[s1.stmt.source] -> stm.subject_label)
           in Policy.ord)
    }
}

-----------------------------------------------------------------
-- Verify no storage covert channels found in current State

assert verify_no_storage_channel {
    all s: State |
       not storage_channel_found[s] }

-- Define conditions under which a storage channel could exist icw
--  a PutDirectFile
pred storage_channel_found [s: State] {
    let stm = s.stmt | {
       stm.type = PutDirectFile &&
       s.direct_file.full = const1 &&
       -- check if direct file was last written by a higher subject
       not ((s.direct_file.last_written -> stm.subject_label)
         in Policy.ord)
    }
}
```

```
-----------------------------------------------------------------------
-- Verify no timing covert channels found in current State

assert verify_no_timing_channel {
    all s: State | not timing_channel_found[s] }

-- Define conditions under which a timing channel could exist
pred timing_channel_found [gc2: State] {
    some disj rw, gc1: State | {
       (gc2 -> rw) in State_order.st_after &&
       (rw -> gc1) in State_order.st_after &&
       gc1.stmt.type = GetClock &&
       gc2.stmt.type = GetClock &&
       rw.stmt.type in (Read_dev + Write_dev
         + PutDirectFile + GetDirectFile) &&
       -- check if GetClocks are at same level
       gc1.stmt.subject_label = gc2.stmt.subject_label &&
       -- check if Read/Write/DirectFile operation at
       --  higher level than GetClock
       not ((rw.stmt.subject_label -> gc2.stmt.subject_label)
         in Policy.ord)
    }
}

-----------------------------------------------------------------------
-- Find a consistent instance of this model
pred show () {}
-----------------------------------------------------------------------
/*******************************/
/** DM Implementation Model **/
/*******************************/
-- The base program is below. Total of 9 statements
-- (S1)   Read_dev ( SysHigh ,  x1 );
-- (S2)   GetClock ( SysLow ,  t1 );
-- (S3)   if ( x1 < 0 )  then
-- (S4)    PutDirectFile ( SysHigh ,  1 , x1 );
-- (S5)   GetClock ( SysLow ,  t2 );
-- (S6)   if ( t1 Before t2 )  then
-- (S7)    Write_dev ( SysLow ,  1 );
--        else
-- (S8)    Write_dev ( SysLow ,  0 );
-- (S9)   Stop;

-------------------------
/*** Statement sigs ***/
-------------------------
one sig S1 extends Statement {}
{
     type = Read_dev
     destination = x1
     source = none
     source_label = none
     key = none
     subject_label =  SysHigh
}

one sig S2 extends Statement {}
{
     type = GetClock
     destination = t1
     source = none
     source_label = none
```

```
     key = none
     subject_label =  SysLow
}

one sig S4 extends Statement {}
{
     type = PutDirectFile
     source = x1
     key = const1
     destination = none
     source_label = none
     subject_label =  SysHigh
}

one sig S3 extends Statement {}
{
     type = Condition
     source =   x1
     destination = none
     source_label = none
     key = none
}

one sig S5 extends Statement {}
{
     type = GetClock
     destination = t2
     source = none
     source_label = none
     key = none
     subject_label =  SysLow
}

one sig S7 extends Statement {}
{
     type = Write_dev
     source = const1
     destination = none
     source_label = none
     key = none
     subject_label =  SysLow
}

one sig S8 extends Statement {}
{
     type = Write_dev
     source = const0
     destination = none
     source_label = none
     key = none
     subject_label =  SysLow
}

one sig S6 extends Statement {}
{
     type = Condition
     source =   t1 + t2
     destination = none
     source_label = none
     key = none
}

one sig S9 extends Statement {}
```

```
{
     type = Stop
     source = none
     destination = none
     source_label = none
     key = none
}

--------------------------------
/*** Variables & Constants ***/
--------------------------------
enum Variable {
     x1, t1, t2
 }

enum Value {
      const_minus_4, const_minus_3, const_minus_2, const0
     , const1, const2, const3, const4

 }

one sig LT {  lt:  Value -> Value }
{  lt = ^(
       (  const_minus_4       ->   const_minus_3)
     + (  const_minus_3       ->   const_minus_2)
     + (  const_minus_2       ->   const0)
     + (  const0   ->   const1)
     + (  const1   ->   const2)
     + (  const2   ->   const3)
     + (  const3   ->   const4)
) }

-------------------------------------
/*** State Transition Predicate ***/
-------------------------------------
fact trans {
     all st1: State - InitialState | some st: State |


     ( st.stmt = S1 &&
       st1.prev_state = st &&
        --  Read_dev
        ( st1.access_label = st.access_label ++ ( x1 ->  SysHigh  ) &&
        some n: Value | st1.vars = st.vars ++ ( x1 -> n) &&
        st1.stmt = S2 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

           -- Part A, copy all dependencies for vars different from x1
           {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x1 }
         )
     ) or

     ( st.stmt = S2 &&
       st1.prev_state = st &&
        --  GetClock
        ( st1.access_label = st.access_label ++ ( t1 ->  SysLow  ) &&
        st1.vars = st.vars ++ ( t1 -> st.current_clock ) &&
        st1.stmt = S3 &&
        st1.direct_file = st.direct_file &&
```

```
        st1.current_clock = st.current_clock &&
        st1.last_cond_checked = st.last_cond_checked
        )
     ) or

     ( st.stmt = S4 &&
       st1.prev_state = st &&
        --  PutDirectFile
        ( st1.stmt = S5 &&
          st1.current_clock = TO/next[st.current_clock] &&
          st1.last_cond_checked = st.last_cond_checked &&
          st1.vars = st.vars &&
          st1.access_label = st.access_label &&
          ( (const1 in st.direct_file.keyContent.Value) =>
             -- the key is found
               (st1.direct_file.success = const1 &&
                st1.direct_file.keyContent = st.direct_file.keyContent
++
                  ( const1 -> st.vars[ x1 ]) &&
                st1.direct_file.keyLabel = st.direct_file.keyLabel ++
                  ( const1 -> SysHigh  ) &&
                -- since key already existed, full remains the same
                st1.direct_file.full = st.direct_file.full
               )
              else -- the key is not found
               ( st.direct_file.full = const0 =>   -- Direct File not
Full
                ( st1.direct_file.keyContent =
st.direct_file.keyContent ++
                  ( const1 -> st.vars[ x1 ]) &&
                  st1.direct_file.keyLabel = st.direct_file.keyLabel
++
                  ( const1 -> SysHigh  ) &&
                  st1.direct_file.success = const1 &&
                  -- if content limit reached, set full to const1
(true)
                  (#st1.direct_file.keyContent =
st1.direct_file.max_slots =>

             st1.direct_file.full = const1 else st1.direct_file.full =
const0)
                )
                else  -- Direct File is Full
                 (st1.direct_file = st.direct_file &&
                  st1.direct_file.success = const0 &&
                  -- assign full to const1 (true)
                  st1.direct_file.full = const1)
            )
          )
        )
     ) or

     ( st.stmt = S3 &&
       st1.prev_state = st &&
        --  if
        ( st1.access_label = st.access_label &&
        st1.vars = st.vars  &&
        st1.current_clock = st.current_clock  &&
        st1.direct_file = st.direct_file  &&
        st1.influenced_by = st.influenced_by &&
        st1.last_cond_checked = {x: st.last_cond_checked | x.stmt !=
S3} + st &&
        (
```

```
          (( st.vars[ x1 ] -> const0) in LT.lt)
           => st1.stmt = S4
           else st1.stmt = S5)
        )
      ) or

      ( st.stmt = S5 &&
        st1.prev_state = st &&
         --  GetClock
         ( st1.access_label = st.access_label ++ ( t2 ->  SysLow  ) &&
         st1.vars = st.vars ++ ( t2 -> st.current_clock ) &&
         st1.stmt = S6 &&
         st1.direct_file = st.direct_file &&
         st1.current_clock = st.current_clock &&
         st1.last_cond_checked = st.last_cond_checked
         )
      ) or

      ( st.stmt = S7 &&
        st1.prev_state = st &&
         --  Write_dev
         ( st1.access_label = st.access_label &&
         st1.stmt = S9 &&
         st1.direct_file = st.direct_file &&
         st1.current_clock = TO/next[st.current_clock] &&
         st1.influenced_by = st.influenced_by &&
         st1.last_cond_checked = st.last_cond_checked
         )
      ) or

      ( st.stmt = S8 &&
        st1.prev_state = st &&
         --  Write_dev
         ( st1.access_label = st.access_label &&
         st1.stmt = S9 &&
         st1.direct_file = st.direct_file &&
         st1.current_clock = TO/next[st.current_clock] &&
         st1.influenced_by = st.influenced_by &&
         st1.last_cond_checked = st.last_cond_checked
         )
      ) or

      ( st.stmt = S6 &&
        st1.prev_state = st &&
         --  if
         ( st1.access_label = st.access_label &&
         st1.vars = st.vars  &&
         st1.current_clock = st.current_clock  &&
         st1.direct_file = st.direct_file  &&
         st1.influenced_by = st.influenced_by &&
         st1.last_cond_checked = {x: st.last_cond_checked | x.stmt !=
S6} + st &&
          (
            (( st.vars[ t1 ] -> st.vars[ t2 ] ) in Clock.before)
             => st1.stmt = S7
             else st1.stmt = S8)
          )
      ) or

      ( st.stmt = S9 &&
        st1.prev_state = st &&
         --  Stop
         ( st1.stmt = st.stmt )
```

```
     )
}
--------------------------
run show for 10 but 32 FTuple
check verify_security for 10 but 32 FTuple
check verify_flow_policy for 10 but 32 FTuple
check verify_no_dependency_flaw for 10 but 32 FTuple
check verify_no_storage_channel for 10 but 32 FTuple
check verify_no_timing_channel for 10 but 32 FTuple
```

APPENDIX B.3 – GENERATED DM FOR BASE PROGRAM EXAMPLE 3

This appendix provides complete code for the trusted subject information flow violation example base program and resultant DM described in Chapter VI - "Example DM Implementations." The DM below is generated by the DM-Compiler from the following base program:

```
(s1)      Read_dev (SysHigh, x1);
(s2)      Read_dev (SysMid, x2);
(s3)      Assign x1 from x2 as SysLow;
(s4)      Write_dev (SysLow, x1);
(s5)      Stop;
```

The Alloy specification for the DM follows:

```
/*****************************************************************/
module static_model
open util/ordering[Time] as TO
/*****************************************************************/

/*************************/
/** DM Invariant Model **/
/*************************/

sig Statement {
    type:  Stmt_type,
    destination: lone Variable,
    source:  set Variable + Value,
    source_label: lone (AccessLabel + Variable),
    key:   lone (Variable + Value),
    subject_label:  lone AccessLabel
}

enum Stmt_type {
    Assign, Condition,
    Read_dev, Write_dev,
    GetDirectFile, PutDirectFile,
    GetClock, Stop
}

-- define access labels based on security policy lattice
enum AccessLabel { SysHigh, SysMid, SysLow }

-- define a Policy signature to allow BLP-style info flows
one sig Policy {
    ord: AccessLabel -> AccessLabel
}
{   ord = ^( (SysLow -> SysMid)
         + (SysMid -> SysHigh) )
         + (iden & (AccessLabel -> AccessLabel) )
}

sig State {
```

```
    stmt:  Statement, -- next stmt to execute
    vars:  Variable -> one (Value + Time), -- variable table
    access_label:  Variable -> one AccessLabel,
    direct_file:  DirectFile, -- current snapshot
    current_clock: Time,
    prev_state: lone State,
    err_msg:  lone Error,
    influenced_by: Variable -> State,
    last_cond_checked: set State,
}
{ -- define error conditions
    ( err_msg = InfoFlow_error <=>
       not consistent_with_FlowPolicy [this] ) &&
    ( err_msg = Overt_flaw_detected <=>
       dependency_flaw_found[this] ) &&
    ( err_msg = Storage_channel_detected <=>
       storage_channel_found[this] ) &&
    ( err_msg = Timing_channel_detected <=>
       timing_channel_found[this] )
}

-- Signature for error types
enum Error {
    InfoFlow_error,
    Overt_flaw_detected,
    Storage_channel_detected,
    Timing_channel_detected
}

-----------------------------------------------------------------
-- Initialization of State signature: all variables initially have 0
--  value and SysLow label, and DirectFile is empty
one sig InitialState extends State {}
{
    vars = (Variable -> const0)
    access_label = (Variable -> SysLow)
    stmt = S1
    direct_file.full = const0
    direct_file.success =const1
    current_clock = TO/first[]
    prev_state =  none
    err_msg =  none
    last_cond_checked = none
    no influenced_by
    no direct_file.keyContent
    no direct_file.keyLabel
}

-- Sig establishes ordering of States in a program execution
one sig State_order {
    st_after: State -> State
}
{   st_after = ^ prev_state
}

-- a "Stop" State cannot precede another State
fact { all s: State | s.prev_state.stmt.type != Stop }

-- no two States can be identical
fact { no disj st1, st2: State |
    (st1.stmt = st2.stmt  &&
     st1.prev_state = st2.prev_state &&
     st1.vars = st2.vars &&
```

```
     st1.direct_file = st2.direct_file)
}

-------------------------------------------------------------------
sig DirectFile {
    -- each key Value is assigned a content Value and AccessLabel
    keyContent:Value -> lone Value,
    keyLabel: Value -> lone AccessLabel,
    last_written: lone AccessLabel,
    full:  (const0 + const1),
    success: (const0 + const1),
    max_slots: Int
}
{      max_slots = 2     -- capacity limited to 2 key locations
}

-------------------------------------------------------------------
sig Time {}

one sig Clock {
    before:  Time -> Time,
    long_before: Time -> Time
}
{ long_before in before &&
    all t1: Time, t2: Time - t1 |
    ((t1->t2) in before <=> t2 in TO/nexts[t1]) &&
    ((t1->t2) in long_before <=> some t3: Time |
       (t3 in before[t1] && t3 in before.t2))
}

-------------------------------------------------------------------
-- Alloy signature used for passing results of tsFilter function
sig FTuple {
    val: Value,
    label: AccessLabel
}

fact { all v: Value, a: AccessLabel | one f: FTuple |
    f.val=v && f.label=a }

-------------------------------------------------------------------
-- Functions, Facts, Assertions and Predicates for info flow security
--  policy and security rules
-------------------------------------------------------------------
-- The tsFilter function defines the semantics of the Trusted Subj
--  Assignment statement, by enabling a TS to act as a Content
--  or Label Filter.
-- Different invariant models may define different filter functions,
--  depending on the TS semantics that must be demonstrated.
fun tsFilter[dv, s1v, s2v: Value,
             da, s1a, s2a: AccessLabel]: FTuple {
{ result: FTuple | {
    result.val = (((s1v->const0) in LT.lt)
        => const0 else s1v)
    result.label = (((da->s2a) in Policy.ord)
        => s2a else
      (((s2a->SysMid) in Policy.ord)
        => SysMid else s2a)) }
} }

-------------------------------------------------------------------
-- Security assertion to verify program abides by all security rules
--  assert verify_security {
```

```
    all s: State |
       consistent_with_FlowPolicy [s] &&
       not dependency_flaw_found [s] &&
       not storage_channel_found [s] &&
       not timing_channel_found [s]
}

-------------------------------------------------------------------
-- Define how statements abide by info flow policy
assert verify_flow_policy {
    all s: State | consistent_with_FlowPolicy[s] }

pred consistent_with_FlowPolicy [s: State] {
    let stm = s.stmt | {
       -- for Write_dev or PutDirectFile statement
       (stm.type in (Write_dev + PutDirectFile) &&
         stm.source in Variable)
       => ((s.access_label[stm.source] -> stm.subject_label)
           in Policy.ord)
    }
}

-------------------------------------------------------------------
-- Verify no overt control dependency flaw found in current State

assert verify_no_dependency_flaw {
    all s: State | not dependency_flaw_found[s] }

-- Define conditions under which a control dependency flaw could
--  exist; checks for a Write, where source in the current state is
--  influenced_by State with higher label in required access.
--  Assertion uses dynamic slicing techniques.
pred dependency_flaw_found [s: State] {
    let stm = s.stmt, s1 = s.influenced_by[stm.source] | {
       stm.type = Write_dev  &&
       stm.source in Variable &&
       -- check if Write_dev source was influenced_by a var
       --  higher than subject
       not ((s1.access_label[s1.stmt.source] -> stm.subject_label)
           in Policy.ord)
    }
}

-------------------------------------------------------------------
-- Verify no storage covert channels found in current State

assert verify_no_storage_channel {
    all s: State |
       not storage_channel_found[s] }

-- Define conditions under which a storage channel could exist icw
--  a PutDirectFile
pred storage_channel_found [s: State] {
    let stm = s.stmt | {
       stm.type = PutDirectFile &&
       s.direct_file.full = const1 &&
       -- check if direct file was last written by a higher subject
       not ((s.direct_file.last_written -> stm.subject_label)
         in Policy.ord)
    }
}

-------------------------------------------------------------------
```

```
-- Verify no timing covert channels found in current State

assert verify_no_timing_channel {
    all s: State | not timing_channel_found[s] }

-- Define conditions under which a timing channel could exist
pred timing_channel_found [gc2: State] {
    some disj rw, gc1: State | {
       (gc2 -> rw) in State_order.st_after &&
       (rw -> gc1) in State_order.st_after &&
       gc1.stmt.type = GetClock &&
       gc2.stmt.type = GetClock &&
       rw.stmt.type in (Read_dev + Write_dev
         + PutDirectFile + GetDirectFile) &&
       -- check if GetClocks are at same level
       gc1.stmt.subject_label = gc2.stmt.subject_label &&
       -- check if Read/Write/DirectFile operation at
       --  higher level than GetClock
       not ((rw.stmt.subject_label -> gc2.stmt.subject_label)
         in Policy.ord)
    }
}

-------------------------------------------------------------------------
-- Find a consistent instance of this model
pred show () {}
-------------------------------------------------------------------------
/*********************************/
/** DM Implementation Model **/
/*********************************/
-- The base program is below. Total of 5 statements
-- (S1)   Read_dev ( SysHigh ,  x1 );
-- (S2)   Read_dev ( SysMid ,  x2 );
-- (S3)   Assign x1 from x2 as SysLow ;
-- (S4)   Write_dev ( SysLow ,  x1 );
-- (S5)   Stop;

-------------------------
/*** Statement sigs ***/
-------------------------
one sig S1 extends Statement {}
{
     type = Read_dev
     destination = x1
     source = none
     source_label = none
     key = none
     subject_label =  SysHigh
}

one sig S2 extends Statement {}
{
     type = Read_dev
     destination = x2
     source = none
     source_label = none
     key = none
     subject_label =  SysMid
}

one sig S3 extends Statement {}
{
     type = Assign
```

```
     source = x2
     destination = x1
     source_label = SysLow
     key = none
}

one sig S4 extends Statement {}
{
     type = Write_dev
     source = x1
     destination = none
     source_label = none
     key = none
     subject_label =  SysLow
}

one sig S5 extends Statement {}
{
     type = Stop
     source = none
     destination = none
     source_label = none
     key = none
}

---------------------------------
/*** Variables & Constants ***/
---------------------------------
enum Variable {
     x1, x2
 }

enum Value {
      const_minus_3, const_minus_2, const0, const1
     , const2
 }

one sig LT {  lt:  Value -> Value }
{  lt = ^(
        (  const_minus_3       ->   const_minus_2)
     + (  const_minus_2        ->   const0)
     + (  const0   ->   const1)
     + (  const1   ->   const2)
) }

-------------------------------------
/*** State Transition Predicate ***/
-------------------------------------
fact trans {
     all st1: State - InitialState | some st: State |


     ( st.stmt = S1 &&
       st1.prev_state = st &&
        --  Read_dev
        ( st1.access_label = st.access_label ++ ( x1 ->  SysHigh  ) &&
        some n: Value | st1.vars = st.vars ++ ( x1 -> n) &&
        st1.stmt = S2 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =
```

```
          -- Part A, copy all dependencies for vars different from x1
          {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x1 }
         )
      ) or

      ( st.stmt = S2 &&
        st1.prev_state = st &&
         --  Read_dev
         ( st1.access_label = st.access_label ++ ( x2 ->  SysMid  ) &&
         some n: Value | st1.vars = st.vars ++ ( x2 -> n) &&
         st1.stmt = S3 &&
         st1.direct_file = st.direct_file &&
         st1.current_clock = TO/next[st.current_clock] &&
         st1.last_cond_checked = st.last_cond_checked &&
         st1.influenced_by =

          -- Part A, copy all dependencies for vars different from x2
          {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x2 }
         )
      ) or

      ( st.stmt = S3 &&
        st1.prev_state = st &&
         --  Trusted Assign
         ( let xx = tsFilter[ st.vars[ x1 ],  st.vars[x2], const0,
           st.access_label[ x1 ],  st.access_label[ x2 ],  SysLow  ] |
(
           st1.vars = st.vars ++ ( x1 -> xx.val ) &&
           st1.access_label = st.access_label ++ ( x1 -> xx.label )
           ) &&
         st1.stmt = S4 &&
         st1.direct_file = st.direct_file &&
         st1.current_clock = st.current_clock &&
         st1.last_cond_checked = st.last_cond_checked &&
         st1.influenced_by =

          -- Part A, copy all dependencies for vars different from x1
          {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x1 }

          -- and inherit all dependencies of the right-hand part x2
          + ( x1 -> st.influenced_by[ x2 ])
         )
      ) or

      ( st.stmt = S4 &&
        st1.prev_state = st &&
         --  Write_dev
         ( st1.access_label = st.access_label &&
         st1.stmt = S5 &&
         st1.direct_file = st.direct_file &&
         st1.current_clock = TO/next[st.current_clock] &&
         st1.influenced_by = st.influenced_by &&
         st1.last_cond_checked = st.last_cond_checked
         )
      ) or

      ( st.stmt = S5 &&
        st1.prev_state = st &&
         --  Stop
         ( st1.stmt = st.stmt )
```

```
     )
}
--------------------------
run show for 6 but 15 FTuple
check verify_security for 6 but 15 FTuple
check verify_flow_policy for 6 but 15 FTuple
check verify_no_dependency_flaw for 6 but 15 FTuple
check verify_no_storage_channel for 6 but 15 FTuple
check verify_no_timing_channel for 6 but 15 FTuple
```

APPENDIX B.4 – GENERATED DM FOR BASE PROGRAM EXAMPLE 4

This appendix provides complete code for the trusted subject information flow with overt dependency flaw violation example base program and resultant DM described in Chapter VI - "Example DM Implementations." The DM below is generated by the DM-Compiler from the following base program:

```
(s1)      Read_dev (SysHigh, x1);
(s2)      Read_dev (SysLow, x2);
(s3)      Read_dev (SysMid, x3);
(s4)      if x1 < 0 then {
(s5)        Assign x1 from x2 as x3;
(s6)        Write_dev (SysMid, x1); }
(s7)      else  Write_dev (SysMid, x1);
(s8)      Stop;
```

The Alloy specification for the DM follows:

```
/*****************************************************************/
module static_model
open util/ordering[Time] as TO
/*****************************************************************/

/*************************/
/** DM Invariant Model **/
/*************************/

sig Statement {
    type:  Stmt_type,
    destination: lone Variable,
    source:  set Variable + Value,
    source_label: lone (AccessLabel + Variable),
    key:   lone (Variable + Value),
    subject_label: lone AccessLabel
}

enum Stmt_type {
    Assign, Condition,
    Read_dev, Write_dev,
    GetDirectFile, PutDirectFile,
    GetClock, Stop
}

-- define access labels based on security policy lattice
enum AccessLabel { SysHigh, SysMid, SysLow }

-- define a Policy signature to allow BLP-style info flows
one sig Policy {
    ord: AccessLabel -> AccessLabel
}
{   ord = ^( (SysLow -> SysMid)
         + (SysMid -> SysHigh) )
         + (iden & (AccessLabel -> AccessLabel) )
```

```
}

sig State {
    stmt:  Statement, -- next stmt to execute
    vars:  Variable -> one (Value + Time), -- variable table
    access_label:  Variable -> one AccessLabel,
    direct_file:  DirectFile, -- current snapshot
    current_clock: Time,
    prev_state: lone State,
    err_msg:  lone Error,
    influenced_by: Variable -> State,
    last_cond_checked: set State,
}
{ -- define error conditions
    ( err_msg = InfoFlow_error <=>
       not consistent_with_FlowPolicy [this] ) &&
    ( err_msg = Overt_flaw_detected <=>
       dependency_flaw_found[this] ) &&
    ( err_msg = Storage_channel_detected <=>
       storage_channel_found[this] ) &&
    ( err_msg = Timing_channel_detected <=>
       timing_channel_found[this] )
}

-- Signature for error types
enum Error {
    InfoFlow_error,
    Overt_flaw_detected,
    Storage_channel_detected,
    Timing_channel_detected
}

-----------------------------------------------------------------------
-- Initialization of State signature: all variables initially have 0
--  value and SysLow label, and DirectFile is empty
one sig InitialState extends State {}
{
    vars = (Variable -> const0)
    access_label = (Variable -> SysLow)
    stmt = S1
    direct_file.full = const0
    direct_file.success =const1
    current_clock = TO/first[]
    prev_state =  none
    err_msg =  none
    last_cond_checked = none
    no influenced_by
    no direct_file.keyContent
    no direct_file.keyLabel
}

-- Sig establishes ordering of States in a program execution
one sig State_order {
    st_after: State -> State
}
{   st_after = ^ prev_state
}

-- a "Stop" State cannot precede another State
fact { all s: State | s.prev_state.stmt.type != Stop }

-- no two States can be identical
fact { no disj st1, st2: State |
```

```
    (st1.stmt = st2.stmt  &&
     st1.prev_state = st2.prev_state &&
     st1.vars = st2.vars &&
     st1.direct_file = st2.direct_file)
}

-------------------------------------------------------------------
sig DirectFile {
    -- each key Value is assigned a content Value and AccessLabel
    keyContent:Value -> lone Value,
    keyLabel: Value -> lone AccessLabel,
    last_written: lone AccessLabel,
    full:  (const0 + const1),
    success: (const0 + const1),
    max_slots: Int
}
{      max_slots = 2     -- capacity limited to 2 key locations
}

-------------------------------------------------------------------
sig Time {}

one sig Clock {
    before:  Time -> Time,
    long_before: Time -> Time
}
{ long_before in before &&
    all t1: Time, t2: Time - t1 |
    ((t1->t2) in before <=> t2 in TO/nexts[t1]) &&
    ((t1->t2) in long_before <=> some t3: Time |
       (t3 in before[t1] && t3 in before.t2))
}

-------------------------------------------------------------------
-- Alloy signature used for passing results of tsFilter function
sig FTuple {
    val:  Value,
    label: AccessLabel
}

fact { all v: Value, a: AccessLabel | one f: FTuple |
    f.val=v && f.label=a }

-------------------------------------------------------------------
-- Functions, Facts, Assertions and Predicates for info flow security
--  policy and security rules
-------------------------------------------------------------------
-- The tsFilter function defines the semantics of the Trusted Subj
--  Assignment statement, by enabling a TS to act as a Content
--  or Label Filter.
-- Different invariant models may define different filter functions,
--  depending on the TS semantics that must be demonstrated.
fun tsFilter[dv, s1v, s2v: Value,
             da, s1a, s2a: AccessLabel]: FTuple {
{ result: FTuple | {
    result.val = (((s1v->const0) in LT.lt)
        => const0 else s1v)
    result.label = (((da->s2a) in Policy.ord)
        => s2a else
      (((s2a->SysMid) in Policy.ord)
        => SysMid else s2a)) }
} }
```

```
-----------------------------------------------------------------------
-- Security assertion to verify program abides by all security rules
--  assert verify_security {
    all s: State |
       consistent_with_FlowPolicy [s] &&
       not dependency_flaw_found [s] &&
       not storage_channel_found [s] &&
       not timing_channel_found [s]
}

-----------------------------------------------------------------------
-- Define how statements abide by info flow policy
assert verify_flow_policy {
    all s: State | consistent_with_FlowPolicy[s] }

pred consistent_with_FlowPolicy [s: State] {
    let stm = s.stmt | {
       -- for Write_dev or PutDirectFile statement
       (stm.type in (Write_dev + PutDirectFile) &&
         stm.source in Variable)
       => ((s.access_label[stm.source] -> stm.subject_label)
           in Policy.ord)
    }
}

-----------------------------------------------------------------------
-- Verify no overt control dependency flaw found in current State

assert verify_no_dependency_flaw {
    all s: State | not dependency_flaw_found[s] }

-- Define conditions under which a control dependency flaw could
--  exist; checks for a Write, where source in the current state is
--  influenced_by State with higher label in required access.
--  Assertion uses dynamic slicing techniques.
pred dependency_flaw_found [s: State] {
    let stm = s.stmt, s1 = s.influenced_by[stm.source] | {
       stm.type = Write_dev  &&
       stm.source in Variable &&
       -- check if Write_dev source was influenced_by a var
       --  higher than subject
       not ((s1.access_label[s1.stmt.source] -> stm.subject_label)
           in Policy.ord)
    }
}

-----------------------------------------------------------------------
-- Verify no storage covert channels found in current State

assert verify_no_storage_channel {
    all s: State |
       not storage_channel_found[s] }

-- Define conditions under which a storage channel could exist icw
--  a PutDirectFile
pred storage_channel_found [s: State] {
    let stm = s.stmt | {
       stm.type = PutDirectFile &&
       s.direct_file.full = const1 &&
       -- check if direct file was last written by a higher subject
       not ((s.direct_file.last_written -> stm.subject_label)
         in Policy.ord)
    }
```

```
}

-----------------------------------------------------------------------
-- Verify no timing covert channels found in current State

assert verify_no_timing_channel {
    all s: State | not timing_channel_found[s] }

-- Define conditions under which a timing channel could exist
pred timing_channel_found [gc2: State] {
    some disj rw, gc1: State | {
       (gc2 -> rw) in State_order.st_after &&
       (rw -> gc1) in State_order.st_after &&
       gc1.stmt.type = GetClock &&
       gc2.stmt.type = GetClock &&
       rw.stmt.type in (Read_dev + Write_dev
         + PutDirectFile + GetDirectFile) &&
       -- check if GetClocks are at same level
       gc1.stmt.subject_label = gc2.stmt.subject_label &&
       -- check if Read/Write/DirectFile operation at
       --  higher level than GetClock
       not ((rw.stmt.subject_label -> gc2.stmt.subject_label)
         in Policy.ord)
    }
}

-----------------------------------------------------------------------
-- Find a consistent instance of this model
pred show () {}
-----------------------------------------------------------------------
/*******************************/
/** DM Implementation Model **/
/*******************************/
-- The base program is below. Total of 8 statements
-- (S1)   Read_dev ( SysHigh ,  x1 );
-- (S2)   Read_dev ( SysLow ,  x2 );
-- (S3)   Read_dev ( SysMid ,  x3 );
-- (S4)   if ( x1 < 0 )  then {
-- (S5)    Assign x1 from x2 as x3 ;
-- (S6)    Write_dev ( SysMid ,  x1 ); }
--        else
-- (S7)    Write_dev ( SysMid ,  x1 );
-- (S8)   Stop;

-------------------------
/*** Statement sigs ***/
-------------------------
one sig S1 extends Statement {}
{
     type = Read_dev
     destination = x1
     source = none
     source_label = none
     key = none
     subject_label =  SysHigh
}

one sig S2 extends Statement {}
{
     type = Read_dev
     destination = x2
     source = none
     source_label = none
```

```
     key = none
     subject_label =  SysLow
}

one sig S3 extends Statement {}
{
     type = Read_dev
     destination = x3
     source = none
     source_label = none
     key = none
     subject_label =  SysMid
}

one sig S5 extends Statement {}
{
     type = Assign
     source = x2
     destination = x1
     source_label = x3
     key = none
}

one sig S6 extends Statement {}
{
     type = Write_dev
     source = x1
     destination = none
     source_label = none
     key = none
     subject_label =  SysMid
}

one sig S7 extends Statement {}
{
     type = Write_dev
     source = x1
     destination = none
     source_label = none
     key = none
     subject_label =  SysMid
}

one sig S4 extends Statement {}
{
     type = Condition
     source =   x1
     destination = none
     source_label = none
     key = none
}

one sig S8 extends Statement {}
{
     type = Stop
     source = none
     destination = none
     source_label = none
     key = none
}

---------------------------------
/*** Variables & Constants ***/
```

```
--------------------------------
enum Variable {
     x1, x2, x3
 }

enum Value {
      const_minus_4, const_minus_3, const_minus_2, const0
     , const1, const2, const3
 }

one sig LT {  lt:  Value -> Value }
{  lt = ^(
       (  const_minus_4       ->   const_minus_3)
     + (  const_minus_3       ->   const_minus_2)
     + (  const_minus_2       ->   const0)
     + (  const0   ->   const1)
     + (  const1   ->   const2)
     + (  const2   ->   const3)
) }

-------------------------------------
/*** State Transition Predicate ***/
-------------------------------------
fact trans {
     all st1: State - InitialState | some st: State |


     ( st.stmt = S1 &&
       st1.prev_state = st &&
        --  Read_dev
        ( st1.access_label = st.access_label ++ ( x1 ->  SysHigh  ) &&
        some n: Value | st1.vars = st.vars ++ ( x1 -> n) &&
        st1.stmt = S2 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

          -- Part A, copy all dependencies for vars different from x1
          {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x1 }
        )
     ) or

     ( st.stmt = S2 &&
       st1.prev_state = st &&
        --  Read_dev
        ( st1.access_label = st.access_label ++ ( x2 ->  SysLow  ) &&
        some n: Value | st1.vars = st.vars ++ ( x2 -> n) &&
        st1.stmt = S3 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

          -- Part A, copy all dependencies for vars different from x2
          {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x2 }
        )
     ) or

     ( st.stmt = S3 &&
       st1.prev_state = st &&
```

```
        --  Read_dev
        ( st1.access_label = st.access_label ++ ( x3 ->  SysMid  ) &&
        some n: Value | st1.vars = st.vars ++ ( x3 -> n) &&
        st1.stmt = S4 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

          -- Part A, copy all dependencies for vars different from x3
          {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x3 }
        )
     ) or

     ( st.stmt = S5 &&
       st1.prev_state = st &&
        --  Trusted Assign
        ( let xx = tsFilter[ st.vars[ x1 ],  st.vars[x2], st.vars[x3],
          st.access_label[ x1 ],  st.access_label[ x2 ],
st.access_label[ x3 ]] | (

          st1.vars = st.vars ++ ( x1 -> xx.val ) &&
          st1.access_label = st.access_label ++ ( x1 -> xx.label )
          ) &&
        st1.stmt = S6 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = st.current_clock &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

          -- Part A, copy all dependencies for vars different from x1
          {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x1 }

          -- and inherit all dependencies of the source_label x3
          + ( x1 -> st.influenced_by[ x3 ])

          -- Part B, all states from last_cond_checked
          -- within which scope this assignment belongs
          + ( x1 -> {x: st.last_cond_checked | x.stmt in  S4} )

          -- Part C, copy dependencies for all variables
participating in
          -- conditions within which scope this assignment belongs
          + ( x1 -> State.{ x: st.last_cond_checked,
            y: x.influenced_by[x.stmt.source] | x.stmt in  S4 } )
        )
     ) or

     ( st.stmt = S6 &&
       st1.prev_state = st &&
        --  Write_dev
        ( st1.access_label = st.access_label &&
        st1.stmt = S8 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.influenced_by = st.influenced_by &&
        st1.last_cond_checked = st.last_cond_checked
        )
     ) or

     ( st.stmt = S7 &&
```

```
        st1.prev_state = st &&
         --  Write_dev
         ( st1.access_label = st.access_label &&
         st1.stmt = S8 &&
         st1.direct_file = st.direct_file &&
         st1.current_clock = TO/next[st.current_clock] &&
         st1.influenced_by = st.influenced_by &&
         st1.last_cond_checked = st.last_cond_checked
         )
      ) or

      ( st.stmt = S4 &&
        st1.prev_state = st &&
         --  if
         ( st1.access_label = st.access_label &&
         st1.vars = st.vars  &&
         st1.current_clock = st.current_clock  &&
         st1.direct_file = st.direct_file  &&
         st1.influenced_by = st.influenced_by &&
         st1.last_cond_checked = {x: st.last_cond_checked | x.stmt !=
S4} + st &&
         (
           (( st.vars[ x1 ] -> const0) in LT.lt)
            => st1.stmt = S5
            else st1.stmt = S7)
         )
      ) or

      ( st.stmt = S8 &&
        st1.prev_state = st &&
         --  Stop
         ( st1.stmt = st.stmt )
      )
}
--------------------------
run show for 9 but 28 FTuple
check verify_security for 9 but 28 FTuple
check verify_flow_policy for 9 but 28 FTuple
check verify_no_dependency_flaw for 9 but 28 FTuple
check verify_no_storage_channel for 9 but 28 FTuple
check verify_no_timing_channel for 9 but 28 FTuple
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B.5 – GENERATED DM FOR BASE PROGRAM EXAMPLE 5

This appendix provides complete code for the trusted subject storage covert channel violation example base program and resultant DM described in Chapter VI - "Example DM Implementations." The DM below is generated by the DM-Compiler from the following base program:

```
(s1)      Read_dev (SysLow, x1);
(s2)      Read_dev (SysLow, x2);
(s3)      Assign x1 from x2 as SysHigh;
(s4)      if x1 > const_minus_1 then {
(s5)         PutDirectFile (SysHigh, 1, x1);
(s6)         PutDirectFile (SysHigh, 2, x2); }
(s7)      PutDirectFile (SysLow, 3, 1);
(s8)      if (full = True) then
(s9)         Write_dev (SysLow, 1);
(s10)     else Write_dev (SysLow, 0);
(s11)     Stop;
```

The Alloy specification for the DM follows:

```
/*****************************************************************/
module static_model
open util/ordering[Time] as TO
/*****************************************************************/

/*************************/
/** DM Invariant Model **/
/*************************/

sig Statement {
    type:  Stmt_type,
    destination: lone Variable,
    source:  set Variable + Value,
    source_label: lone (AccessLabel + Variable),
    key:   lone (Variable + Value),
    subject_label:  lone AccessLabel
}

enum Stmt_type {
    Assign, Condition,
    Read_dev, Write_dev,
    GetDirectFile, PutDirectFile,
    GetClock, Stop
}

-- define access labels based on security policy lattice
enum AccessLabel { SysHigh, SysMid, SysLow }

-- define a Policy signature to allow BLP-style info flows
one sig Policy {
    ord: AccessLabel -> AccessLabel
}
```

```
{    ord = ^( (SysLow -> SysMid)
          + (SysMid -> SysHigh) )
          + (iden & (AccessLabel -> AccessLabel) )
}

sig State {
    stmt:  Statement, -- next stmt to execute
    vars:  Variable -> one (Value + Time), -- variable table
    access_label:  Variable -> one AccessLabel,
    direct_file:  DirectFile, -- current snapshot
    current_clock: Time,
    prev_state: lone State,
    err_msg:  lone Error,
    influenced_by: Variable -> State,
    last_cond_checked: set State,
}
{ -- define error conditions
    ( err_msg = InfoFlow_error <=>
       not consistent_with_FlowPolicy [this] ) &&
    ( err_msg = Overt_flaw_detected <=>
       dependency_flaw_found[this] ) &&
    ( err_msg = Storage_channel_detected <=>
       storage_channel_found[this] ) &&
    ( err_msg = Timing_channel_detected <=>
       timing_channel_found[this] )
}

-- Signature for error types
enum Error {
    InfoFlow_error,
    Overt_flaw_detected,
    Storage_channel_detected,
    Timing_channel_detected
}

-------------------------------------------------------------------
-- Initialization of State signature: all variables initially have 0
--  value and SysLow label, and DirectFile is empty
one sig InitialState extends State {}
{
    vars = (Variable -> const0)
    access_label = (Variable -> SysLow)
    stmt = S1
    direct_file.full = const0
    direct_file.success =const1
    current_clock = TO/first[]
    prev_state =  none
    err_msg =  none
    last_cond_checked = none
    no influenced_by
    no direct_file.keyContent
    no direct_file.keyLabel
}

-- Sig establishes ordering of States in a program execution
one sig State_order {
    st_after: State -> State
}
{    st_after = ^ prev_state
}

-- a "Stop" State cannot precede another State
fact { all s: State | s.prev_state.stmt.type != Stop }
```

```
-- no two States can be identical
fact { no disj st1, st2: State |
    (st1.stmt = st2.stmt  &&
     st1.prev_state = st2.prev_state &&
     st1.vars = st2.vars &&
     st1.direct_file = st2.direct_file)
}

-------------------------------------------------------------------
sig DirectFile {
    -- each key Value is assigned a content Value and AccessLabel
    keyContent:Value -> lone Value,
    keyLabel: Value -> lone AccessLabel,
    last_written: lone AccessLabel,
    full:  (const0 + const1),
    success: (const0 + const1),
    max_slots: Int
}
{      max_slots = 2     -- capacity limited to 2 key locations
}

-------------------------------------------------------------------
sig Time {}

one sig Clock {
    before:  Time -> Time,
    long_before: Time -> Time
}
{ long_before in before &&
    all t1: Time, t2: Time - t1 |
    ((t1->t2) in before <=> t2 in TO/nexts[t1]) &&
    ((t1->t2) in long_before <=> some t3: Time |
       (t3 in before[t1] && t3 in before.t2))
}

-------------------------------------------------------------------
-- Alloy signature used for passing results of tsFilter function
sig FTuple {
    val: Value,
    label: AccessLabel
}

fact { all v: Value, a: AccessLabel | one f: FTuple |
    f.val=v && f.label=a }

-------------------------------------------------------------------
-- Functions, Facts, Assertions and Predicates for info flow security
--  policy and security rules
-------------------------------------------------------------------
-- The tsFilter function defines the semantics of the Trusted Subj
--  Assignment statement, by enabling a TS to act as a Content
--  or Label Filter.
-- Different invariant models may define different filter functions,
--  depending on the TS semantics that must be demonstrated.
fun tsFilter[dv, s1v, s2v: Value,
             da, s1a, s2a: AccessLabel]: FTuple {
{ result: FTuple | {
    result.val = (((s1v->const0) in LT.lt)
        => const0 else s1v)
    result.label = (((da->s2a) in Policy.ord)
        => s2a else
      (((s2a->SysMid) in Policy.ord)
```

```
        => SysMid else s2a)) }
} }

-------------------------------------------------------------------
-- Security assertion to verify program abides by all security rules
--  assert verify_security {
    all s: State |
       consistent_with_FlowPolicy [s] &&
       not dependency_flaw_found [s] &&
       not storage_channel_found [s] &&
       not timing_channel_found [s]
}

-------------------------------------------------------------------
-- Define how statements abide by info flow policy
assert verify_flow_policy {
    all s: State | consistent_with_FlowPolicy[s] }

pred consistent_with_FlowPolicy [s: State] {
    let stm = s.stmt | {
       -- for Write_dev or PutDirectFile statement
       (stm.type in (Write_dev + PutDirectFile) &&
         stm.source in Variable)
       => ((s.access_label[stm.source] -> stm.subject_label)
          in Policy.ord)
    }
}

-------------------------------------------------------------------
-- Verify no overt control dependency flaw found in current State

assert verify_no_dependency_flaw {
    all s: State | not dependency_flaw_found[s] }

-- Define conditions under which a control dependency flaw could
--  exist; checks for a Write, where source in the current state is
--  influenced_by State with higher label in required access.
--  Assertion uses dynamic slicing techniques.
pred dependency_flaw_found [s: State] {
    let stm = s.stmt, s1 = s.influenced_by[stm.source] | {
       stm.type = Write_dev  &&
       stm.source in Variable &&
       -- check if Write_dev source was influenced_by a var
       --  higher than subject
       not ((s1.access_label[s1.stmt.source] -> stm.subject_label)
           in Policy.ord)
    }
}

-------------------------------------------------------------------
-- Verify no storage covert channels found in current State

assert verify_no_storage_channel {
    all s: State |
       not storage_channel_found[s] }

-- Define conditions under which a storage channel could exist icw
--  a PutDirectFile
pred storage_channel_found [s: State] {
    let stm = s.stmt | {
       stm.type = PutDirectFile &&
       s.direct_file.full = const1 &&
       -- check if direct file was last written by a higher subject
```

```
       not ((s.direct_file.last_written -> stm.subject_label)
         in Policy.ord)
    }
}

-----------------------------------------------------------------------
-- Verify no timing covert channels found in current State

assert verify_no_timing_channel {
    all s: State | not timing_channel_found[s] }

-- Define conditions under which a timing channel could exist
pred timing_channel_found [gc2: State] {
    some disj rw, gc1: State | {
       (gc2 -> rw) in State_order.st_after &&
       (rw -> gc1) in State_order.st_after &&
       gc1.stmt.type = GetClock &&
       gc2.stmt.type = GetClock &&
       rw.stmt.type in (Read_dev + Write_dev
         + PutDirectFile + GetDirectFile) &&
       -- check if GetClocks are at same level
       gc1.stmt.subject_label = gc2.stmt.subject_label &&
       -- check if Read/Write/DirectFile operation at
       --  higher level than GetClock
       not ((rw.stmt.subject_label -> gc2.stmt.subject_label)
         in Policy.ord)
    }
}

-----------------------------------------------------------------------
-- Find a consistent instance of this model
pred show () {}
-----------------------------------------------------------------------
/*******************************/
/** DM Implementation Model **/
/*******************************/
-- The base program is below. Total of 11 statements
-- (S1)   Read_dev ( SysLow ,  x1 );
-- (S2)   Read_dev ( SysLow ,  x2 );
-- (S3)   Assign x1 from x2 as SysHigh ;
-- (S4)   if ( x1 > const_minus_1 )  then {
-- (S5)    PutDirectFile ( SysHigh ,  1 , x1 );
-- (S6)    PutDirectFile ( SysHigh ,  2 , x2 ); }
-- (S7)   PutDirectFile ( SysLow ,  3 , 1 );
-- (S8)   if ( full = True )  then
-- (S9)    Write_dev ( SysLow ,  1 );
--        else
-- (S10)   Write_dev ( SysLow ,  0 );
-- (S11)  Stop;

-------------------------
/*** Statement sigs ***/
-------------------------
one sig S1 extends Statement {}
{
     type = Read_dev
     destination = x1
     source = none
     source_label = none
     key = none
     subject_label =  SysLow
}
```

```
one sig S2 extends Statement {}
{
     type = Read_dev
     destination = x2
     source = none
     source_label = none
     key = none
     subject_label =  SysLow
}

one sig S3 extends Statement {}
{
     type = Assign
     source = x2
     destination = x1
     source_label = SysHigh
     key = none
}

one sig S5 extends Statement {}
{
     type = PutDirectFile
     source = x1
     key = const1
     destination = none
     source_label = none
     subject_label =  SysHigh
}

one sig S6 extends Statement {}
{
     type = PutDirectFile
     source = x2
     key = const2
     destination = none
     source_label = none
     subject_label =  SysHigh
}

one sig S4 extends Statement {}
{
     type = Condition
     source =   x1 + const_minus_1
     destination = none
     source_label = none
     key = none
}

one sig S7 extends Statement {}
{
     type = PutDirectFile
     source = const1
     key = const3
     destination = none
     source_label = none
     subject_label =  SysLow
}

one sig S9 extends Statement {}
{
     type = Write_dev
     source = const1
     destination = none
```

```
     source_label = none
     key = none
     subject_label =  SysLow
}

one sig S10 extends Statement {}
{
     type = Write_dev
     source = const0
     destination = none
     source_label = none
     key = none
     subject_label =  SysLow
}

one sig S8 extends Statement {}
{
     type = Condition
     source = none
     destination = none
     source_label = none
     key = none
}

one sig S11 extends Statement {}
{
     type = Stop
     source = none
     destination = none
     source_label = none
     key = none
}

---------------------------------
/*** Variables & Constants ***/
---------------------------------
enum Variable {
     x1, x2, const_minus_1
 }

enum Value {
      const_minus_4, const_minus_3, const_minus_2, const0
     , const1, const2, const3, const4
     , const5, const6
 }

one sig LT {  lt:  Value -> Value }
{  lt = ^(
       (  const_minus_4       ->   const_minus_3)
     + (  const_minus_3       ->   const_minus_2)
     + (  const_minus_2       ->   const0)
     + (  const0   ->   const1)
     + (  const1   ->   const2)
     + (  const2   ->   const3)
     + (  const3   ->   const4)
     + (  const4   ->   const5)
     + (  const5   ->   const6)
) }

-----------------------------------
/*** State Transition Predicate ***/
-----------------------------------
fact trans {
```

```
     all st1: State - InitialState | some st: State |


     ( st.stmt = S1 &&
       st1.prev_state = st &&
        --  Read_dev
        ( st1.access_label = st.access_label ++ ( x1 ->  SysLow  ) &&
        some n: Value | st1.vars = st.vars ++ ( x1 -> n) &&
        st1.stmt = S2 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

           -- Part A, copy all dependencies for vars different from x1
           {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x1 }
         )
     ) or

     ( st.stmt = S2 &&
       st1.prev_state = st &&
        --  Read_dev
        ( st1.access_label = st.access_label ++ ( x2 ->  SysLow  ) &&
        some n: Value | st1.vars = st.vars ++ ( x2 -> n) &&
        st1.stmt = S3 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

           -- Part A, copy all dependencies for vars different from x2
           {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x2 }
         )
     ) or

     ( st.stmt = S3 &&
       st1.prev_state = st &&
        --  Trusted Assign
        ( let xx = tsFilter[ st.vars[ x1 ],  st.vars[x2], const0,
          st.access_label[ x1 ],  st.access_label[ x2 ],  SysHigh  ]
| (
          st1.vars = st.vars ++ ( x1 -> xx.val ) &&
          st1.access_label = st.access_label ++ ( x1 -> xx.label )
          ) &&
        st1.stmt = S4 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = st.current_clock &&
        st1.last_cond_checked = st.last_cond_checked &&
        st1.influenced_by =

           -- Part A, copy all dependencies for vars different from x1
           {v: Variable, s: State | (v -> s) in st.influenced_by &&
v!= x1 }

           -- and inherit all dependencies of the right-hand part x2
           + ( x1 -> st.influenced_by[ x2 ])
        )
     ) or

     ( st.stmt = S5 &&
       st1.prev_state = st &&
```

```
        --  PutDirectFile
        ( st1.stmt = S6 &&
          st1.current_clock = TO/next[st.current_clock] &&
          st1.last_cond_checked = st.last_cond_checked &&
          st1.vars = st.vars &&
          st1.access_label = st.access_label &&
          ( (const1 in st.direct_file.keyContent.Value) =>
             -- the key is found
               (st1.direct_file.success = const1 &&
                st1.direct_file.keyContent = st.direct_file.keyContent
++
                  ( const1 -> st.vars[ x1 ]) &&
                st1.direct_file.keyLabel = st.direct_file.keyLabel ++
                  ( const1 -> SysHigh  ) &&
                -- since key already existed, full remains the same
                st1.direct_file.full = st.direct_file.full
               )
              else -- the key is not found
               ( st.direct_file.full = const0 =>   -- Direct File not
Full
                ( st1.direct_file.keyContent =
st.direct_file.keyContent ++
                  ( const1 -> st.vars[ x1 ]) &&
                  st1.direct_file.keyLabel = st.direct_file.keyLabel
++
                  ( const1 -> SysHigh  ) &&
                  st1.direct_file.success = const1 &&
                  -- if content limit reached, set full to const1
(true)
                  (#st1.direct_file.keyContent =
st1.direct_file.max_slots =>

             st1.direct_file.full = const1 else st1.direct_file.full =
const0)
                )
                else  -- Direct File is Full
                 (st1.direct_file = st.direct_file &&
                  st1.direct_file.success = const0 &&
                  -- assign full to const1 (true)
                  st1.direct_file.full = const1)
               )
           )
         )
     ) or

     ( st.stmt = S6 &&
       st1.prev_state = st &&
        --  PutDirectFile
        ( st1.stmt = S7 &&
          st1.current_clock = TO/next[st.current_clock] &&
          st1.last_cond_checked = st.last_cond_checked &&
          st1.vars = st.vars &&
          st1.access_label = st.access_label &&
          ( (const2 in st.direct_file.keyContent.Value) =>
             -- the key is found
               (st1.direct_file.success = const1 &&
                st1.direct_file.keyContent = st.direct_file.keyContent
++
                  ( const2 -> st.vars[ x2 ]) &&
                st1.direct_file.keyLabel = st.direct_file.keyLabel ++
                  ( const2 -> SysHigh  ) &&
                -- since key already existed, full remains the same
                st1.direct_file.full = st.direct_file.full
```

```
              )
             else -- the key is not found
              ( st.direct_file.full = const0 =>   -- Direct File not
Full
               ( st1.direct_file.keyContent =
st.direct_file.keyContent ++
                  ( const2 -> st.vars[ x2 ]) &&
                  st1.direct_file.keyLabel = st.direct_file.keyLabel
++
                  ( const2 -> SysHigh  ) &&
                  st1.direct_file.success = const1 &&
                  -- if content limit reached, set full to const1
(true)
                  (#st1.direct_file.keyContent =
st1.direct_file.max_slots =>

             st1.direct_file.full = const1 else st1.direct_file.full =
const0)
                )
                else  -- Direct File is Full
                 (st1.direct_file = st.direct_file &&
                  st1.direct_file.success = const0 &&
                  -- assign full to const1 (true)
                  st1.direct_file.full = const1)
               )
           )
         )
     ) or

     ( st.stmt = S4 &&
       st1.prev_state = st &&
        --  if
        ( st1.access_label = st.access_label &&
        st1.vars = st.vars  &&
        st1.current_clock = st.current_clock  &&
        st1.direct_file = st.direct_file  &&
        st1.influenced_by = st.influenced_by &&
        st1.last_cond_checked = {x: st.last_cond_checked | x.stmt !=
S4} + st &&
        (
          (( st.vars[ const_minus_1 ] -> st.vars[ x1 ] ) in LT.lt)
           => st1.stmt = S5
           else st1.stmt = S7)
        )
     ) or

     ( st.stmt = S7 &&
       st1.prev_state = st &&
        --  PutDirectFile
        ( st1.stmt = S8 &&
          st1.current_clock = TO/next[st.current_clock] &&
          st1.last_cond_checked = st.last_cond_checked &&
          st1.vars = st.vars &&
          st1.access_label = st.access_label &&
          ( (const3 in st.direct_file.keyContent.Value) =>
             -- the key is found
               (st1.direct_file.success = const1 &&
                st1.direct_file.keyContent = st.direct_file.keyContent
++
                  ( const3 -> const1) &&
                st1.direct_file.keyLabel = st.direct_file.keyLabel ++
                  ( const3 -> SysLow  ) &&
                -- since key already existed, full remains the same
```

```
                st1.direct_file.full = st.direct_file.full
               )
              else -- the key is not found
               ( st.direct_file.full = const0 =>   -- Direct File not
Full
                ( st1.direct_file.keyContent =
st.direct_file.keyContent ++
                  ( const3 -> const1) &&
                  st1.direct_file.keyLabel = st.direct_file.keyLabel
++
                  ( const3 -> SysLow  ) &&
                  st1.direct_file.success = const1 &&
                  -- if content limit reached, set full to const1
(true)
                  (#st1.direct_file.keyContent =
st1.direct_file.max_slots =>

             st1.direct_file.full = const1 else st1.direct_file.full =
const0)
                 )
                 else  -- Direct File is Full
                  (st1.direct_file = st.direct_file &&
                   st1.direct_file.success = const0 &&
                   -- assign full to const1 (true)
                   st1.direct_file.full = const1)
               )
           )
         )
     ) or

     ( st.stmt = S9 &&
       st1.prev_state = st &&
        --  Write_dev
        ( st1.access_label = st.access_label &&
        st1.stmt = S11 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.influenced_by = st.influenced_by &&
        st1.last_cond_checked = st.last_cond_checked
        )
     ) or

     ( st.stmt = S10 &&
       st1.prev_state = st &&
        --  Write_dev
        ( st1.access_label = st.access_label &&
        st1.stmt = S11 &&
        st1.direct_file = st.direct_file &&
        st1.current_clock = TO/next[st.current_clock] &&
        st1.influenced_by = st.influenced_by &&
        st1.last_cond_checked = st.last_cond_checked
        )
     ) or

     ( st.stmt = S8 &&
       st1.prev_state = st &&
        --  if
        ( st1.access_label = st.access_label &&
        st1.vars = st.vars  &&
        st1.current_clock = st.current_clock  &&
        st1.direct_file = st.direct_file  &&
        st1.influenced_by = st.influenced_by &&
```

```
        st1.last_cond_checked = {x: st.last_cond_checked | x.stmt !=
S8} + st &&
        ( ( st.direct_file.full = const1 )
           => st1.stmt = S9
           else st1.stmt = S10)
        )
     ) or

     ( st.stmt = S11 &&
       st1.prev_state = st &&
        --  Stop
        ( st1.stmt = st.stmt )
     )
}
--------------------------
run show for 12 but 40 FTuple
check verify_security for 12 but 40 FTuple
check verify_flow_policy for 12 but 40 FTuple
check verify_no_dependency_flaw for 12 but 40 FTuple
check verify_no_storage_channel for 12 but 40 FTuple
check verify_no_timing_channel for 12 but 40 FTuple
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Research Office, Code 09
   Naval Postgraduate School
   Monterey, CA

4. Dr. Mikhail Auguston
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA

5. Dr. Cynthia Irvine
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA

6. Dr. Gurminder Singh
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA

7. Dr. Gordon Bradley
   Operations Research Department
   Naval Postgraduate School
   Monterey, CA

8. Timothy Levin
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA

9. Dr. Dennis Volpano
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA

10. Dr. Bret Michael
Computer Science Department
Naval Postgraduate School
Monterey, CA

11. CDR Alan B. Shaffer, USN
Computer Science Department
Naval Postgraduate School
Monterey, CA

12. Dr. Ralph Wachter
Office of Naval Research
Arlington, VA

13. Karl Levitt
National Science Foundation
Arlington, VA

14. Dr. John Monastra
National Reconnaissance Office
Chantilly, VA

15. Dr. Daniel Jackson
MIT Computer Science and AI Lab
The Strata Center
Cambridge, MA

16. Dr. Jin Song Dong
Computer Science Department, School of Computing
National University of Singapore
Singapore