DENNING, Dorothy Elizabeth Robling, 1945-
SECURE INFORMATION FLOW IN COMPUTER SYSTEMS.

Purdue University, Ph.D., 1975
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48106

SECURE INFORMATION FLOW IN COMPUTER SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Dorothy Elizabeth Robling Denning

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 1975

Grad. School
Form No. 9
Revised

PURDUE UNIVERSITY

Graduate School

This is to certify that the thesis prepared

By Dorothy Elizabeth Robling Denning

Entitled Secure Information Flow in Computer Systems

Complies with the University regulations and that it meets the accepted standards of the Graduate School with respect to originality and quality

For the degree of:

Doctor of Philosophy

Signed by the final examining committee:

Herbert D. Schwetman, chairman

Saul Rosen

Kenneth Omahen

Approved by the head of school or department:

May 1 1975

To the librarian:

is
This thesis [✓ is not] to be regarded as confidential

Herbert D. Schwetman
Professor in charge of the thesis

ACKNOWLEDGMENTS

I am grateful to Herbert D. Schwetman for his assistance in the preparation of this thesis, and to Kenneth J. Omahen, Saul Rosen, and Andrew B. Whinston for their advice and support. I wish to thank Carl Ellison and Butler Lampson for their suggestions on handling program interrupts. I am indebted to my husband, Peter J. Denning, for his criticisms, suggestions, patience, and constant encouragement.

I wish also to acknowledge IBM for awarding me a fellowship, and the National Science Foundation for their support through grants GJ-43176 and GJ-41289.

TABLE OF CONTENTS

CHAPTER Page

3 (cont.)

LIST OF FIGURES

Figure Page

LIST OF SYMBOLS

Symbol	Meaning	Page
$\rightarrow$	Flow relation and partial ordering relation on the security classes SC	16
$\Rightarrow$	Derives relation on storage objects, security classes, and items of information	29
$\oplus$	Class-combining operator and least upper bound operator on pairs of security classes of SC	16
$\bigoplus$	Least upper bound operator on sets of security classes of SC	42
$\otimes$	Greatest lower bound operator on pairs of security classes of SC	43
$\bigotimes$	Greatest lower bound operator on sets of security classes of SC	44
$\underline{\quad}$	Operator denoting the security class of an element	38
$\prec$	Partial ordering relation on the processes P	69
A,B	Security classes of SC	16
a,b	Logical storage objects of N	19
$\underline{a}$	Security class of object a	38
$(\underline{a})_i$	Security class of object a in state s_i	38
AM	Access control matrix	21
c	Class sequence	71
D_p	Domain (input set) of process p	69
d	Structure index of a program	133
F	Set of files	50
FM	Information flow model	16

Symbol	Meaning	Page
FS	Information flow structure	16
f	Function or operation	17
H	Highest security class of SC	45
h	Table of upper bounds on Security Classes	137
HS	Hardware stack of security classes	123
$\underline{HS}$	Security class on top of HS	123
IFD	Immediate forward dominator block	97
L	Lowest security class of SC	39
ℓ	Table of lower bounds on security classes	137
M	Set of physical memory locations	19
N	Set of logical storage objects ($N = F \cup W$)	19
P	Set of processes	19
p,q	Processes of P	19
[p	Initiation of process p	69
p]	Termination of process p	69
R_p	Range (output set) of process p	69
r	Process initiation or termination event	71
S	Set of states	16
s	State of S	19
SS	Software stack of (low,high) pairs of security classes	137
$\underline{SS}.\ell$	Low component of security class pair on top of SS	137
$\underline{SS}.h$	High component of security class pair on top of SS	137
T	Set of transition operators	16
t	Transition operator of T	27
u,v	Physical storage locations of M	19

Symbol	Meaning	Page
V	Set of objects assigned values in a program	62
W	Working store or computational store	50
W_p	Working store of process p	51
α,β	Items of information	27
δ	Maximum path length in partial order graph	144
λ	Empty sequence	71
ν	Final value of a class sequence	72
ξ	Expression	61
Π	Process system	69
ρ	Execution sequence of a process system	71
σ	State sequence corresponding to an execution sequence	71
τ	Transition operator sequence	27
Φ	Program or abstract program structure	61
Ψ	Set of basic blocks	97
ω	String of terminal and non-terminal symbols	86

ABSTRACT

Denning, Dorothy Elizabeth Robling. Ph.D., Purdue University, May 1975. Secure Information Flow in Computer Systems. Major Professor: Herbert D. Schwetman.

This thesis investigates mechanisms that guarantee the secure flow of information between a set of security classes in a computing system. These mechanisms are examined within the framework of a mathematical model suitable for concisely formulating the requirements of secure information flow. The model provides a unifying view of all systems that restrict information flow and enables a classification of them according to security objectives.

The central component of the model is a lattice structure derived from the security classes and justified by the semantics of information flow. The lattice properties are used to develop two mechanisms that verify the secure execution of a program addressing storage locations associated with different security classes. The first operates in an environment in which the security class associated with each location is invariant. The second operates in a more complex environment in which the security class associated with each location is to be determined by the information contained in it.

INTRODUCTION

Chapter 1

This thesis is a theoretical investigation of mechanisms that guarantee secure information flow in a computer system. By information flow we mean that information in one object (e.g., a variable or file) is used to derive information in another. By secure information flow, or simple security, we mean that no unauthorized flow of information is possible. The mechanisms that guarantee the security of a system are referred to as protection mechanisms.

We restrict our study to the special case where the flow of information is to be constrained by a "flow relation" on a set of "security class". We develop a mathematical model of information flow suitable for formulating the security requirements of such a system. The properties of the model are used to construct automatic program certification mechanisms that verify the secure execution of a program.

In the remainder of this introduction we discuss informally the security problem we wish specifically to solve and the approach followed thereafter to its solution. (See [Po74,Hf69] for a survey of other security problems.)

1.1 Problem Statement

The problem under investigation may be informally stated as follows: given a set of "security classes" corresponding to classes of information, and a specification of allowable paths by which information can flow among them, construct a mechanism which guarantees that the flows which arise during program executions do not violate the specification. The security classes are intended to model the important concepts of "security classifications", "security categories", and "need to know" [We69,Ga72]; as will be seen, however, they are more general in their scope. Information is considered to flow from one class to another if it flows from an object associated with the former to one associated with the latter.

Consider for example a system, in which all information is partitioned into four security levels: A_1, A_2, A_3, A_4, where $A_1 < A_2 < A_3 < A_4$. The security requirement of this system is that information may flow from A_i to A_j only if $i \leq j$. Interpreting $A_i < A_j$ to mean that "A_i is at a lower security level than A_j," this requirement states simply that high security information cannot flow into low security classes. In a government or military system, the classes $A_1,\ldots,A_4$ could correspond to unclassified, classified, secret, and top secret information respectively. In this case, the security requirement is to prevent events such as top secret data's from entering an unclassified file or reaching a user with only a secret clearance. This partially describes the security requirements of ADEPT-50, a time sharing system designed to handle sensitive information in government and military systems [We69].

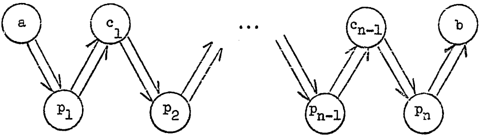
To guarantee security in such a system, it is necessary to block all communication channels or "flow paths" between objects in class A_i and class A_j whenever $A_i > A_j$. For instance, consider two files a and b belonging to classes A_i and A_j respectively. The security requirement of the system implies that data may be transferred from a to b if and only if $A_i \leq A_j$. According to this requirement, transferring data from a top secret file a to an unclassified file b is not permitted.

A major implementation problem with this type of security requirement is detecting and monitoring all possible flow paths from a to b; most direct paths are simply detected, but indirect paths — involving one or more other processes and files — are considerably more difficult to detect. The problem can be stated precisely as follows. For a file a and process p, let a => p denote the proposition "p can read a into its local computational memory," and p => a denote the proposition "p can write from its local computational memory into a." Then let a => b mean there exists a process p such that a => p => b, and $a \overset{*}{=>} b$ mean there exist processes $p_1,\dots,p_n$ and files $c_0,\dots,c_n$ $(n \geq 1)$ such that

$$a = c_0 => p_1 => c_1 => p_2 => \dots => p_{n-1} => c_{n-1} => p_n => c_n = b.$$

When n = 1 the flow $a \overset{*}{=>} b$ is "direct", and when n > 1 it is "indirect". The foregoing security requirement is now stated simply: no flow $a \overset{*}{=>} b$ for $a \in A_i$ and $b \in A_j$ is permissible unless $A_i \leq A_j$.

Figure 1.1-1 illustrates two representations of the flow $a \overset{*}{=>} b$: a graphical form and an "access matrix" form. The latter form was first introduced by Lampson [La71] and developed by Graham and



Graphical Representation

Files

	a	c_1	c_2	...	c_{n-2}	c_{n-1}	b
p_1	r	w					
p_2		r	w				
Processes ⋮							
p_{n-1}					r	w	
p_n						r	w

Access Matrix Representation

Figure 1.1-1. Communication Channel from a to b.

Denning [GD72]. It has proved to be a useful conceptual basis within which to study problems of access control [Jo73]. However, since it does not include notions of security classes and information flow, one of our tasks will be extending it so that, for example, granting a read or write access to a process is not permitted if it introduces an insecure flow path.

The example above is easily extended to allow for processes that communicate directly between their computational memories (for example, using "send message" and "get message" operations [Ha70]) by regarding some of the c_i as message buffers rather than files.

The example is of course a rather simple instance of a more general problem. The complications which we shall study include, but are not limited to:

1. The security classes A_i do not conform to a strict linear ordering relation.
2. The processes involved in effecting a flow $a \overset{*}{\Rightarrow} b$ are themselves associated with security classes — e.g., according to the classes of their owning users — and these classes may vary during execution. Process security classes must be accounted for in deciding whether a flow $a \overset{*}{\Rightarrow} b$ is permissible.
3. The classes with which files are associated may be allowed to change during a computation. Similarly, the security level of a process's local computational memory may change according to the highest class of information residing within it.

4. Programs, or the systems on which they operate, must be certified as to whether they meet the stated security requirements before they are permitted to execute.

Even if adequate solutions for controlling information flow among specified security classes are instituted, there remain many subtle ways information can flow in violation of the stated security requirements. Some of these troublesome flows utilize the mechanisms put into place to control the direct and indirect flows cited above! Indeed, were it not for the so-called "covert channels" along which information can flow, and were it not for the difficulty in identifying such channels, protection might not be the enormous problem we find today [La73]. A famous example of a covert path is a process that alters its demands on some system resource in a pattern (e.g., Morse Code) that can be observed by an accomplice process; though the capacity of such a channel may be at best a few bits per second, this may be quite sufficient for a patient perpetrator. Another example is a process that discloses a confidential value x by opening x files, again under the observation of an accomplice. Other examples can be constructed around the principle of using error-checking mechanisms to divulge information by the very existence of an error; we shall see specific cases of this later. These problems have been perplexing researchers for many years (see, for example, Lampson [La73], Rotenberg [Ro74, and Fenton [Fe74a]). The mechanisms developed in this thesis provide solutions to many of them.

It is important to note that the protection requirements we have specified here are stronger than the read-write access control requirements found in many contemporary systems, such as MULTICS [Or72,SS72] and HYDRA [Wu74], in that indirect flow paths are not controlled in these latter systems. For example, in MULTICS, if a process p is given read-access to a file a, it may be able to communicate, via a second file b, the entire contents of a to another process p' that was denied read-access to a. More precisely, existence of a path $a \Rightarrow p$ may imply the existence of a path $a \overset{*}{\Rightarrow} p'$ (namely $a \Rightarrow p \Rightarrow b \Rightarrow p'$), even though p' was not permitted to establish a direct path $a \Rightarrow p'$. Similarly, in HYDRA, if a process is given a "capability" to read a file, it can pass this capability to another process (provided the capability has the "copy" option). This does not imply that the protection mechanisms of MULTICS or HYDRA are insecure; it implies only that they are inadequate for the problem under investigation in this thesis. It is also important to realize that the access control mechanisms of these systems handle additional security requirements that we are not explicitly considering here. For example, the mechanisms of both HYDRA and MULTICS permit the controlled execution of code segments; the mechanism for creating new types of objects in HYDRA permits greater control over the use of files. Here, for instance, the "owner" of a file can require that access to it be only through a specified set of procedures.

1.2 Previous Research

The problem under investigation has also been studied by Weissman [We69], Rotenberg [Ro74], Walter et. al. [Wa74], Bell and LaPadula [BL73a,BL73b], and Fenton [Fe73,Fe74a,Fe74b]. Weissman describes the ADEPT-50 system (mentioned earlier), in which control over information flow is based on a set theoretic model of access rights. The security classes (or profiles, as they are called) are sets derived from two properties, Authority and Category, corresponding respectively to government and military Classification and Compartments. Each user, terminal, program, and file is assigned a profile. The basic set operations (e.g., union, intersection) are used to control access to objects and the flow of information. Rotenberg proposes the use of "privacy restrictions" for controlling information flow in a large time sharing system. Here, the security classes are sets of restrictions. A Privacy Restriction Processor records the flow of information in the system by propagating restrictions among the restriction sets associated with segments and processes. The restriction sets are then used to prevent unauthorized releases of information. Walter et. al. report on an investigation at Case Western Reserve to design a Security Kernel that monitors the flow of information between files and processes in a system like MULTICS. Their approach has been to develop a series of three models, M_0, M_1, M_2. M_0 represents the most abstract model and contains agents and repositories which must obey four basic axioms of information flow. Structured modeling techniques are then used to arrive at consistent, but successively less abstract models M_1 and M_2. They are currently investigating implementation of the final model M_2. Bell and LaPadula describe a project at MITRE to develop a set theoretic model of protection for government and military systems. Like ADEPT and the CASE system, their model is intended for systems that control information flow between files and processes. Fenton examines the problem of controlling information flow within a program; i.e., controlling the flow of information between the storage registers accessible to a program. He proposes a restricted hardware design in which each storage register is tagged according to its information class and the instruction execution mechanism inhibits instructions that would produce information flows not permitted by the tags.

We shall return to these works, examining them in more detail in the light of the general model of information flow to be developed in Chapter 2. The objective of this thesis is to extend and improve these research efforts. The model presented in the next chapter gives a formal basis for proving that security requirements are met. By providing a general enough context, it gives a single medium of discussion for comparing and contrasting the five systems above and others as well. The semantics of the security problem impose a well-developed structure on the permissible flows among security classes. This structure is exploited to construct efficient verification and checking algorithms.

The problem of certifying security properties has also been previously investigated by Bell and Burke [BB74], Neumann and Fabry [NF74], Popek and Kline [PK74] and Walter et. al. [Wa74]. However, their research efforts have been directed primarily toward the problem of manually certifying the correct operation and implementation of a complete system, whereas our efforts have been directed exclusively toward the problem of automatically certifying the secure execution of a program in an otherwise secure system.

1.3 Organization and Results of Thesis

In Chapter 2 we introduce a formal model of information flow and state a definition of security in terms of the model. An examination of the mathematical properties of the model shows that under certain assumptions the set of security classes forms a lattice. These assumptions are not arbitrarily chosen, but rather follow from the semantics of the problem. The lattice structure is exploited in later chapters to develop efficient security certification algorithms. Special cases of the model are examined in terms of their security requirements and relation to previous work. Finally, the problem of guaranteeing "determinacy" of security classes in a multiprogrammed environment is analyzed.

In Chapters 3 and 4 we investigate the special case of guaranteeing the secure flow of information within a program written in a well-structured high level language, such as PASCAL [Wr71]. In particular, we present algorithms for proving (or disproving) certain security properties for an arbitrary program. The significance of these algorithms is that the secure execution of a program can be established prior to its execution by a certification procedure. In Chapter 3 we present an algorithm for certifying the secure execution of a program in an environment in which the security class of each storage register remains constant throughout the lifetime of the program. Because it exploits the lattice structure of the security classes, the certification procedure can easily be embedded into the analysis phase of a compiler without adding substantially to the compilation time of a program. In Chapter 4 we propose a hardware design, based on tagged architecture, for dynamically binding registers to security classes, so that the class of a register can be a function of its contents. We then present two possible algorithms for certifying the secure execution of a program in this environment.

In Chapter 5 we state the conclusions reached by this research and propose areas for future study. The important contributions of this research are summarized below:

1. A formal model of information flow and security is developed. It is used for stating and proving the security and determinacy requirements of systems that control information flow.
2. Analysis of the properties of the model, and of the security problem itself, leads to the conclusion that it is semantically reasonable to assume that the set of security classes forms a lattice. The lattice, which appears frequently in studies of programming languages and other semantic problems in computer science, is found to be important in the security problem as well. That lattices are inherent in this problem is the principal reason that verification of security requirements is not only feasible — it is understandable and efficient.
3. The model is a generalization of previous work on the security problem. It gives a unifying view of this work, enables a classification of it according to security objectives, and suggests some new approaches.
4. The model enables the specification of compile-time algorithms for certifying the secure execution of a program, one in an

environment in which the security class of each storage register is invariant, the other in the more complex environment in which the security class of some (or all) storage registers can change in accordance with the class of the information contained in them.

5. The model allows the study of run-time verification mechanisms, leading to the conclusion that such mechanisms are, in most cases, insufficient to enforce properly the security requirements of a system. Certain forms of certification must be performed prior to program execution (point 4, above) to guarantee the correct use of run-time mechanisms.

A MODEL OF INFORMATION FLOW

Chapter 2

In Section 1.1 we observed that the access control mechanisms of most systems do not control the flow of information in a system. Instead they monitor only a process's immediate read and write access rights (or privileges) to objects. In these systems, whether or not information will flow correctly is an undecidable question [HR74]. Consequently the security of such general systems relies greatly on trust.

The objective of this research is to eliminate this element of undecidability and reduce the need for trust by finding suitable constraints under which the security of a system can be decided and enforced. Our results have been more than encouraging: not only do such constraints exist, but they are rich in mathematical structure.

In Section 2.1 we introduce a model suitable for formulating the concept of information flow and specifying the requirements of secure information flow. The model consists of three components: an information flow structure, states, and transition operators. The information flow structure consists of a set of security classes A, B, ... representing different classes of information; a binary relation $\rightarrow$, where $A \rightarrow B$ means that information associated with class A is permitted to flow to class B; and a binary operator $\oplus$, where $A \oplus B$ represents the class associated with information derived from classes A and B (e.g., the result of an arithmetic operation performed on information in classes A and B). The states consist of logical storage objects, an address mapping function from these objects to physical memory, processes, functions that bind objects and processes to security classes, and relations specifying processes' read and write access rights to objects. The transition operators consist of all operations that can cause a change of state.

In Section 2.2 the security requirements of the model are stated: no set of processes must be able to effect a flow of information from class A into class B unless A → B. This flow could occur, for example, as the result of one or more processes transferring (via a sequence of read and write operations) information form a storage object assigned to class A to one assigned to class B. The analysis reveals that security can be guaranteed if each operation is independently secure when the flow relation → is transitive.

An investigation of the mathematical properties of the model in Section 2.3 shows that under certain assumptions, justified by the semantics of information flow, the security classes form a lattice. Specifically, we show the semantics of information flow imply that we are justified in assuming that a) the security classes are partially ordered by the flow relation →, b) ⊕ is a least upper bound operator, and c) there exists a greatest lower bound operator, which we denote by ⊗. This result permits concise formulations of the security requirements of different systems and facilitates the construction of mechanisms that enforce these requirements.

In Section 2.4 special cases of the model are classified along two dimensions: static v. dynamic binding and single v. multi-class. "Static binding" means that the security class associated with an object remains constant over the lifetime of the system; at the opposite extreme, "dynamic binding" means that the security class of an object is a function of its contents and therefore varies with its contents. The difference between static and dynamic binding is illustrated with a simple example. Suppose a process p wishes to move information from its working store (i.e., computational memory) W_p to a file b. Under static binding, it is necessary to verify that the flow from W_p to b is valid exactly once, at the time p opens b for writing. Under dynamic binding, it is necessary to update the class of b by that of W_p each time p writes into b.

A "single-class" working store of a process is treated as a single object bound to a single security class; in contrast, a "multi-class" working store of a process is treated as more than one object and allows each object to be bound to a different class. Security in a single-class system depends only on file input and output operations and therefore can be verified at low cost. In contrast, security in a multi-class system may depend on every instruction executed by a process and therefore is inherently more costly to verify. All of the known mechanisms for controlling information flow are characterized by these two dimensions.

A determinacy problem arises in dynamically bound systems if the security class of an object depends on the order in which a set of processes execute. Since this may be undesirable in some systems, in

Section 2.5 we state additional constraints that guarantee determinate operation with respect to security classes.

2.1 Description of the Model

An information flow model FM is defined by

FM = <FS, S, T>

where

FS is an information flow structure.

S is a set of states.

T is a set of transition operators.

The information flow structure FS models the valid flow paths of a system and is defined by

FS = $\langle SC, \rightarrow, \oplus \rangle$

where

SC = $\{A, B, \ldots\}$ is a set of security classes closed under $\oplus$.

$\rightarrow \subseteq SC \times SC$ is a binary flow relation.

$\oplus$: $SC \times SC \rightarrow SC$ is a binary class-combining operator.

The security classes SC correspond to disjoint classes of information. The flow relation → on SC × SC is defined by $A \rightarrow B$ if and only if information in class A is permitted to "flow" into class B. (A full discussion of the implications of "flow" comes later). If $A \rightarrow B$, we say that A is a lower security class than B, or equivalently that B is a higher security class than A. If there exists a class A such that $A \rightarrow B$ for all $B \in SC$, then A is said to be a lowest security class. If there exists a class B such that $A \rightarrow B$ for all $A \in SC$, then B is said to be a highest security class. In Section 2.3 we shall show that the semantics of the problem imply we are justified to assume the existence of unique lowest and highest security classes.

The class-combining operator $\oplus$ specifies the class that corresponds to the combination of information in two classes (e.g., by addition, multiplication, or insertion into a common file); that is, $A \oplus B$ is the class corresponding to the result of an operation f applied to information in classes A and B. In Section 2.3 we shall show that the semantics of the problem imply we are justified in assuming that the $\oplus$ operator is a least upper bound operator on the set of classes SC.

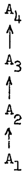
Figure 2.1-1 shows two examples of information flow structures on four security classes, where the flow relation → is a partial ordering of the classes; i.e., → is reflexive, transitive, and anti-symmetric. Example (a) shows the special case of a linear ordering, with a lowest security class A_1 and a highest security class A_4; it corresponds to the example described in Chapter 1. Example (b) shows a non-linear (partial) ordering, with a lowest security class A_{00} and a highest security class A_{11}. The graphical representations are standard precedence graphs for a partial order, showing only the non-reflexive, immediate (irredundant) relations; the remaining relations are implied by transitivity (i.e., a full partial order graph is the transitive closure of its precedence graph). In Section 2.3 we shall show that the semantics of the problem imply we are justified in assuming that the flow relation always partially orders the security classes.

Although we have defined an information flow model FM in terms of a single flow structure FS, an actual system may require more than one flow structure. For example, several disjoint flow structures may be needed to specify the valid flows between different types of objects.

$SC = \{A_1, A_2, A_3, A_4\}$

$A_i \rightarrow A_j$ iff $i \leq j$

$A_i \oplus A_j \equiv A_{\max(i,j)}$



Description

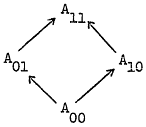
Representation

a) A Linear Ordering

$SC = \{A_{00}, A_{11}, A_{10}, A_{11}\}$

$A_{ij} \rightarrow A_{i'j'}$ iff $i \leq i'$, $j \leq j'$

$A_{ij} \oplus A_{i'j'} \equiv A_{\max(i,i'),\max(j,j')}$



Description

Representation

b) A Non-Linear (Partial) Ordering

Figure 2.1-1. Examples of Partially Ordered Information Flow Structures.

In this case it is a simple matter to extend the model to include a set of flow structures $\{FS_1, \ldots, FS_n\}$.

The states S model the state-dependent components of a protection system. These are the set of active processes, the set of logical memory objects (i.e., the receptacles of information), the binding of objects to security classes, the binding (if any) of processes to security classes, the current configuration of processes' read and write access rights (i.e., configuration of the access matrix), the binding of physical memory locations to objects and classes, and in general all information pertaining to the "security state" of the system. For each state $s \in S$, the following vector is sufficient for our purposes here:

$$s = \langle N, M, P, \underline{loc}, \underline{class}, \underline{tag}, \underline{clearance}, \underline{read}, \underline{write} \rangle$$

where

$N = \{a, b, \ldots\}$ is a set of logical memory objects.

$M = \{u, v, \ldots\}$ is a set of physical memory locations.

$P = \{p, q, \ldots\}$ is a set of processes.

$\underline{loc}: N \rightarrow \underline{powerset}(M)$ is a mapping function from logical memory objects to subsets of physical memory.

$\underline{class}: N \rightarrow SC$ is a function that binds each object in logical memory to a security class.

$\underline{tag}: M \rightarrow SC$ is a function that binds each location in physical memory to a security class.

$\underline{clearance}: P \rightarrow SC$ is a function that binds each process to a security class.

$\underline{read} \subseteq P \times N$ is a relation specifying processes' read access rights (or privileges) to objects.

write $\subseteq$ P × N — is a relation specifying processes' write access rights (or privileges) to objects.

The logical memory N is the set of logical storage objects (i.e., the information receptacles) in the system. We refer to the elements of N simply as objects. Objects may be files, segments, or even program variables, depending on the level of detail under consideration. The physical memory M is the set of physical storage locations in the system. Some elements of M may be located in the main memory, others on secondary storage devices such as disks or drums. Each location u in M represents the smallest unit of memory that can be assigned to a logical object (see the description of loc below). For example, in a paged virtual memory system, u may be a single page in main memory or a sector of a track on drum. We assume that all physical memory locations are addressed only through logical memory objects. The processes P are the active agents responsible for all information flow in the system.

The function loc: N -> powerset(M) assigns one or more (usually contiguous) physical storage locations to each object; that is, for $a \in N$, loc(a) = U, where $U \subseteq M$. For example, if a is a segment in a paged virtual memory system, U would be the set of pages in main memory and the set of track sectors in secondary memory occupied by a.

The functions class: N -> SC, tag: M -> SC, and clearance: P -> SC respectively bind each object, physical location, and process to a security class. The tag function is so named to suggest the use of hardware tags for binding physical memory locations to classes. Although tagged architectures [Fu73] present a possible direct implementation, tags can also be implicit, as fields in page or segment tables or in file descriptor tables. Moreover, the latter form (file tags) are easily added to most existing file systems. However, it is necessary to make explicit the requirement that the implementation of security classes must conform to the logical structure; i.e., for object a, we require that $\underline{\text{tag}}(u) = \underline{\text{class}}(a)$ for all $u \in \underline{\text{loc}}(a)$. If location u has contents α and tag A (i.e., $\underline{\text{tag}}(u) = A$), this will be illustrated by

$$\underset{u}{\boxed{\;A\;\big|\;\alpha\;}}\ .$$

We assume that all objects are logically and, therefore, physically disjoint; that is, $\underline{\text{loc}}(a) \cap \underline{\text{loc}}(b) = \emptyset$ for all $a,b \in N$. Without this assumption, for any a,b not independent (i.e., $a \cap b \neq \emptyset$ and therefore $\underline{\text{loc}}(a) \cap \underline{\text{loc}}(b) \neq \emptyset$), several complications arise when information flows in or out of $\underline{\text{loc}}(a) \cap \underline{\text{loc}}(b)$. For example, if information flows into object a, but is stored in $\underline{\text{loc}}(a) \cap \underline{\text{loc}}(b)$, it must guaranteed that it is permitted to flow into b as well as a. Or, if the class of a is changed, causing a corresponding change to $\underline{\text{tag}}(u)$ for all $u \in \underline{\text{loc}}(a)$, then the class of b should also change.

The relations $\underline{\text{read}}$ and $\underline{\text{write}}$ on P × N specify processes' read and write access rights (or privileges) to objects. These relations are equivalent to the well known $|P| \times |N|$"access control matrix" AM defined by

$r \in AM[p,a]$ iff $p\ \underline{\text{read}}\ a$;

$w \in AM[p,a]$ iff $p\ \underline{\text{write}}\ a$.

(See Figure 1.1-1 for an example of an access matrix and [La71,GD72] for a thorough treatment of the use of an access matrix as a model of protection.) Of course, if a process has read access to an object a and write access to an object b (i.e., p <u>read</u> a and p <u>write</u> b), this does not necessarily mean that it can exercise these privileges and transfer information from a to b; the information transfer must also be permitted by the flow relation (i.e., <u>class</u>(a) → <u>class</u>(b)).

The <u>read</u> and <u>write</u> relations need not be implemented as a matrix. Some methods that are commonly used include "capabilities", "access control lists", and "locks and keys". The capability method corresponds to storing the matrix by rows. Associated with each process p is a set of pairs

$$CL = \{(a, AM[p,a]) \mid AM[p,a] \neq \emptyset\}.$$

The elements of the set are referred to as "capabilities" and the set CL as a "capability list". Capabilities were first introduced by Dennis and VanHorn [DV66] and their utility is becoming increasingly apparent [An74,EL72,En72,Fa68,Fa71,Fa74,Il68,Jo73,Li73,Ne72,St74,Wu74]. The access control method corresponds to storing the matrix by columns. Associated with each object a is a set of pairs

$$ACL = \{(p, AM[p,a]) \mid AM[p,a] \neq \emptyset\}$$

referred to as an "access control list" [SS72]. The lock and key method is a combination of the capability and access control list methods and is commonly used to control access to physical memory [IB68]. Popek examines methods for finding optimum representations of access control relationships expressed with locks and keys [Po73]. For a comparison of the relative costs and reliability aspects of these methods see [De74,GD72,Po74,Ts73,Wi72].

When we need to identify the components of a particular state s_i to distinguish them from those of another state s_j, we shall subscript the components with i: N_i, $class_i$, tag_i, etc..

The transition operators T model all important operations that affect or effect the flow of information in the system. For example, granting a process p write access to an object a affects the flow of information if p may subsequently move information into a; executing an assignment operation "b := a" effects the flow of information by causing data to be transferred from a to b (actually from loc(a) to loc(b)). The operations under consideration here are given in Figure 2.1-2. For each operation, a brief description is given together with the conditions necessary for execution of the operation (i.e., pre-conditions on the state variables) and the conditions that result from its execution (i.e., post-conditions on the state variables). We assume that no changes are made to state variables except as explicitly stated.

The operation transfer($f,a_1,\ldots,a_n,b$) represents any action that causes $f(a_1,\ldots,a_n)$ to flow to b. For example, execution of the statements "b := a", "b := a1 + a2", and "output a to b" will respectively be represented by transfer(:=,a,b), transfer(+,a1,a2,b), and transfer(output,a,b). Note that the operation transfer($f,a_1,\ldots,a_n,b$) does not necessarily replace the entire contents of loc(b) with $f(a_1,\ldots,a_n)$; it may instead extend or modify it by $f(a_1,\ldots,a_n)$ (as, for example, with an output operation to a file b). We shall assume that all transfer operations to an object b replace the contents of loc(b) when b is a single element (e.g., a scalar variable) and modify it when b is an aggregate of elements (e.g., a file or segment).

Operation	Description	Pre-Conditions	Post-Conditions
transfer$(f,a_1,\ldots,a_n,b)$	An n-ary operation f is performed on $a_1,\ldots,a_n$ and the result stored in b.	$a_1,\ldots,a_n,b\varepsilon N$; $\exists p\varepsilon P$: p read a_i $(1\leq i\leq n)$ & p write b	class(b) and tag(u) $\forall u\varepsilon$loc(b) may change
create object(a,A,U)	Object a is created with class A and assigned to storage locations U.	$a\varepsilon N$; $\forall b\varepsilon N$: loc(b)$\cap U=\emptyset$	$a\varepsilon N$; loc(a) = U; class(a) = A; tag(u) = A $\forall u\varepsilon U$
delete object(a)	Object a is deleted.	$a\varepsilon N$	$a\not\varepsilon N$
move(a,U)	Object a is moved to locations U.	$a\varepsilon N$; loc(a)=V; $\forall b\varepsilon N$: loc(b)$\cap U=\emptyset$	loc(a)=U; tag(u)=class(a) $\forall u\varepsilon U$

reclassify(a,A)	Class of a is changed to A.	$a\varepsilon N$	class(a)=A;
			tag(u)=A $\forall u\varepsilon$loc(a)

clear(u)	Location u is cleared.	$\nexists b\varepsilon N$: $u\varepsilon$loc(b)	tag(u) may change
create process(p,A)	Process p is created with a clearance of A.	$p\not\varepsilon P$	$p\varepsilon P$; clearance(p)=A
delete process(p)	Process p is deleted.	$p\varepsilon P$	$p\not\varepsilon P$
grant$(p,a,\{\begin{smallmatrix}r\\w\\rw\end{smallmatrix}\})$	Process p is granted read and/or write access to a.	$p\varepsilon P$; $a\varepsilon N$	p read a and/or p write a
ungrant$(p,a,\{\begin{smallmatrix}r\\w\\rw\end{smallmatrix}\})$	Process p loses read and/or write access to a.	$p\varepsilon P$; $a\varepsilon N$	$\neg$ p read a and/or p write a

Figure 2.1-2. Transition Operators.

The operation *create object*(a,A,U) creates a new logical object a in class A, and assigns it to (unoccupied) storage locations U. The tag of each element $u \in U$ is also set to A, however, the contents of u is not initialized. Note that the operation *create object*(a,A,U) does not automatically give the process that executed it read or write access to the object; this must be acquired by subsequent execution of a *grant* operation (see below). This is done so that all changes to the "access matrix" are modeled by just two operations (*grant* and *ungrant*). The operation *delete object*(a) removes object a, releasing its physical memory locations.

The operation *move*(a,U) assigns new (unoccupied) storage locations U to object a, while transferring the contents of the old locations V to U. The tag of each element $u \in U$ is also set to *class*(a). The *move* operation would be performed, for example, by a "storage compacter" or by a relocation mechanism in a virtual memory system. It is easy to see that the pre-condition "$\forall b \in N$: *loc*$(b) \cap U = \emptyset$" on the operations *create*(a,A,U) and *move*(a,U) guarantees that no two objects will ever share physical memory, since these are the only two operations that assign storage to objects.

The operation *reclassify*(a,A) changes the class of object a and the tag of its corresponding storage locations, without changing its contents. This operation might be useful in a government or military system, where it is necessary to change the classification of data.

The operation *clear*(u) clears (unoccupied) location u (e.g., by filling with zeros or an undefined value). The tag of u is then set to the security class associated with unoccupied storage locations.

The operation create process(p,A) creates a new process p with clearance A. The class A could, for example, correspond to the security clearance of the user on whose behalf p will execute. Sometimes there will exist processes for which no clearance is specified or even necessary; in such cases the class A associated with p would be undefined. Note that the create process operation does not allocate any storage to p; this would have to be done by execution of create object and grant operations. The delete process(p) operation simply removes process p from the system.

The grant and ungrant operations add and remove elements of the read and write relations (i.e., change entries in the access matrix). For example, the operation grant(p,a,r) could be performed when a process p requests that a file a be opened for input (i.e., reading). In this case, granting p this access privilege may depend on the clearance of p, what other processes have access to a, or other state variables not explicitly accounted for in our model. In some systems the controlled granting and ungranting of access privileges may be coordinated with the security requirements — i.e., so that all accesses granted are consistent with permissible flows. In other systems the granting and ungranting of access privileges may not be coordinated with the security requirements; in this case, a process may be denied a request to write into a file to which it has an access privilege if this would create a disallowed flow path.

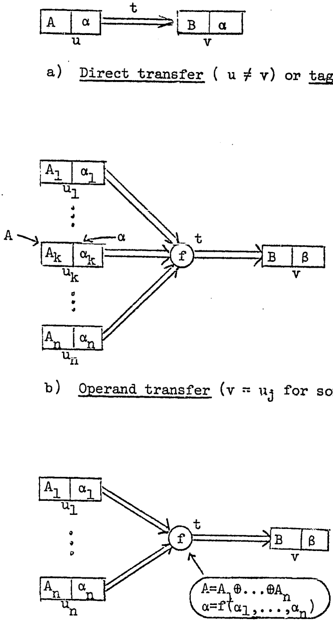
2.2 Security Requirements

The security requirements of the model are simply stated: all information flow must satisfy the constraints of the flow structure; that is, information in class A must not flow into class B unless A → B. A precise formulation of the meaning of "flow" can be given in terms of sequences of state transition operators. For state s and sequence $\tau = t_1 \cdot \ldots \cdot t_m$ of transition operators, we say that τ is feasible from state s if it can be executed beginning in state s (i.e., the preconditions for each t_i $(1 \leq i \leq m)$ are satisfied); in this case we write $s \cdot \tau$ to denote the successor state of s under τ. Two observations are important: 1) in any state, all information in the system is stored in some physical location, and 2) new information is derived only by applying a function f to existing information (in our terms, by some process executing an operation transfer$(f,a_1,\ldots,a_n,b)$).

Now, suppose the system is in state s_i and some process executes an operation t that causes a transition to state s_j (i.e., $s_i \cdot t = s_j$). Let α denote an item of information and let $\alpha \in A$ mean that α is stored in a location u such that tag(u) = A. Then information can flow from class A to class B as a result of a single operation t if and only if there exists an information item $\alpha \in A$ in s_i that derives an information item $\beta \in B$ in s_j ($\alpha = \beta$ is possible). This can occur as the result of one of three possibilities (see Figure 2.2-1):

a) In s_i, α is stored in location u for which $\text{tag}_i(u) = A$, and in s_j, α is stored in location v for which $\text{tag}_j(v) = B$ (v = u is possible).

b) There exist locations $u_1,\ldots,u_n$ $(n \geq 1)$ such that: the



a) <u>Direct transfer</u> ($u \neq v$) or <u>tag change</u> ($u = v$).

b) <u>Operand transfer</u> ($v = u_j$ for some u_j is possible).

c) <u>Function value transfer</u>($v = u_j$ for some u_j is possible).

Figure 2.2-1. Information Flow from A to B by an Operation t.

contents of u_k is α_k and $\underline{\text{tag}}_i(u_k) = A_k$ $(1 \leq k \leq n)$, and the result $\beta = f(\alpha_1,\ldots,\alpha_n)$ of some function f applied to the contents of locations $u_1,\ldots,u_n$ is stored in a location v for which $\underline{\text{tag}}_j(v) = B$. Then $\alpha = \alpha_k$ and $A = A_k$ for some k.

c) Same conditions as case (b), except that $\alpha = \beta = f(\alpha_1,\ldots,\alpha_n)$ and $A = A_1 \oplus \ldots \oplus A_n$. (This extended use of the $\oplus$ operator is justified in the next section.)

Note that case (a) corresponds to a transfer between locations $u \neq v$ or a tag change in a given location $u = v$; case (b) corresponds to the given value's being an input operand to a function which influences the result place in location v; case (c) corresponds to the given value's being generated functionally, in a class to which none of the operands may belong, and then being placed in location v.

We will use the notation $\alpha \overset{s\cdot t}{\Rightarrow} \beta$ to denote the derivation of information β in s•t from information α in s as the result of executing operation t in s, and $A \overset{s\cdot t}{\Rightarrow} B$ to denote the flow of information from class A to class B as the result of executing t in s. Clearly $A \overset{s\cdot t}{\Rightarrow} B$ if and only if there exists an $\alpha \in A$ in s and a $\beta \in B$ in s•t such that $\alpha \overset{s\cdot t}{\Rightarrow} \beta$. Since a flow $A \overset{s.t}{\Rightarrow} A$ occurs trivially whenever an operation is performed that does not affect a location u (i.e., $\underline{\text{tag}}(u) = A$ before and after), we consider only flows $A \overset{s\cdot t}{\Rightarrow} B$ for $A \neq B$.

Consider two states s_i and s_j such that $s_i \cdot t = s_j$ for some t. It is easy to see that for <u>delete object</u>, <u>clear</u>, <u>create process</u>, <u>delete process</u>, <u>grant</u>, or <u>ungrant</u> there is no flow of information. For t = <u>move</u>(a,U), information cannot change classes since the contents of a's storage locations are moved to U, replacing the former contents of u and changing its tag. Hence, any information item $\alpha \in \underline{\text{loc}}_i(a)$ is moved to $U = \underline{\text{loc}}_j(a)$ without changing class (the previous information $\beta \in U$ is lost). This leaves just the three operations: *transfer*, *create object*, and *reclassify* to consider as flow-causing operations. For $t = \underline{\text{transfer}}(f, a_1,\ldots,a_n,b)$, $A \overset{s_i\cdot t}{\Rightarrow} B$ whenever $\underline{\text{class}}_i(a_k) = A$ for some a_k $(1 \le k \le n)$ and/or $\underline{\text{class}}_i(a_1) \oplus \ldots \oplus \underline{\text{class}}_i(a_n) = A$, and $\underline{\text{class}}_j(b) = B$. This is illustrated in Figure 2.2-2 (a). For $t = \underline{\text{create object}}(b,B,U)$, $A \overset{s_i\cdot t}{\Rightarrow} B$ whenever $\underline{\text{tag}}_i(u) = A$ for some $u \in U$, indicating that it was previously occupied by an object a in class A (which has now been deleted or moved elsewhere), but that its contents have not been cleared. Hence, the information in u flows from A to B since the *create object* operation sets $\underline{\text{tag}}_j(u) = B$. This type of flow occurs frequently in systems that do not clear storage before assigning it to new objects. It is illustrated in Figure 2.2-2 (b). For $t = \underline{\text{reclassify}}(a,B)$, $A \overset{s_i\cdot t}{\Rightarrow} B$ whenever $\underline{\text{class}}_i(a) = A$, since *reclassify* sets $\underline{\text{class}}_j(a) = B$ and $\underline{\text{tag}}_j(u) = B$ for all $u \in \underline{\text{loc}}_i(a)$, thus also changing the class of information in $\underline{\text{loc}}_i(a)$. Figure 2.2-2 (c) illustrates this type of flow.

Information can also flow from a class A to a class B as the result of execution of a sequence of operations $\tau = t_1\cdot t_2\cdot\ldots\cdot t_m$. For this case, let $\alpha \overset{s\cdot\tau}{\Rightarrow} \beta$ denote the derivation of information item β in $s\cdot\tau$ from information item α in s as the result of execution of τ beginning in s. Clearly $\alpha \overset{s\cdot\tau}{\Rightarrow} \beta$ implies there exists information items $\gamma_0,\ldots,\gamma_m$ (not necessarily distinct), such that

$$\alpha = \gamma_0 \overset{s.t_1}{\Rightarrow} \gamma_1 \overset{s_1\cdot t_2}{\Rightarrow} \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{m-1} \overset{s_{m-1}\cdot t_m}{\Rightarrow} \gamma_m = \beta,$$

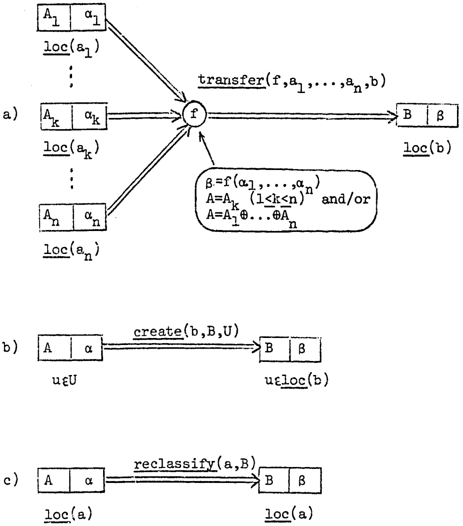


Figure 2.2-2. Operations that Cause Information Flow from A to B.

where $s_1,\ldots,s_m$ is the sequence of successor states of s. Also, let $A \overset{s\cdot\tau}{\Rightarrow} B$ denote the flow of information from A to B as the result of execution of the sequence τ beginning in state s. Then it is also clear that $A \overset{s_i\cdot\tau}{\Rightarrow} B$ if and only if there exists an item α in s_i stored in location u such that $\underline{\text{tag}}_i(u) = A$ and an item β in $s_j = s_i \cdot \tau$ stored in location v such that $\underline{\text{tag}}_j(v) = B$, and $\alpha \overset{s_i\cdot\tau}{\Rightarrow} \beta$. Examples of operation sequences τ that cause information flow from A to B are shown in Figure 2.2-3.

The security requirements of the model are now precisely stated: An operation sequence τ is a secure sequence if and only if for every initial state s such that τ is feasible (if any), $A \overset{s\cdot\tau}{\Rightarrow} B$ implies $A \rightarrow B$. A flow model FM is secure if and only if every possible operation sequence is secure.

To verify the security of a particular system, it is necessary first to show that the system fits the model and then to show that each possible operation sequence $\tau = t_1 \cdot t_2 \cdot \ldots \cdot t_m$ is secure. For m = 1, this involves showing that the operations transfer, create object, and reclassify cannot alone cause a flow from A to B when $A \not\rightarrow B$. Furthermore, if the system is designed to always clear memory before allocating it to a new object, then it is sufficient to establish the secure execution of the operations transfer and reclassify. To see how this can be done, let s_i be an initial state and let $s_j = s_i \cdot t_1$. For $t_1 = \underline{\text{reclassify}}(a,B)$, security can be established by verifying that $\underline{\text{class}}_i(a) \rightarrow B$. Note that this implies that any operation that "declassifies" data (e.g., by changing it from "secret" to "unclassified") is considered to be insecure. This does not mean that such operations must

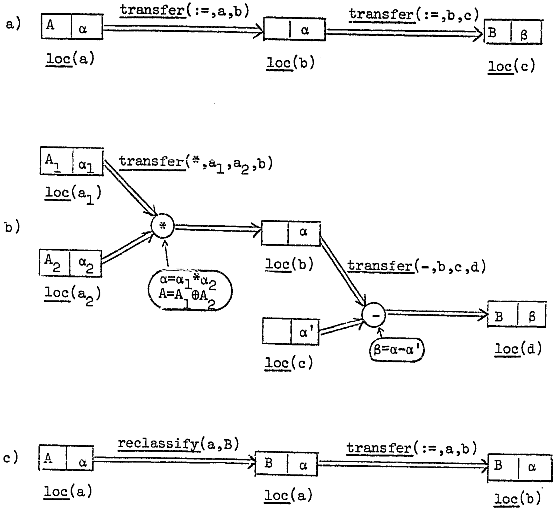


Figure 2.2-3. Examples of Operation Sequences that Cause Information Flow from A to B.

never be allowed; but it does mean that if such operations are necessary, they must be regarded as a potential threat to the security of the system and should be carefully controlled.

For m > 1, verifying the security of a sequence $t_1 \cdot t_2 \cdot \ldots \cdot t_m$ is much more difficult, since a prefix of it may be secure, while the entire sequence may not be. For example, suppose the flow relation satisfies the properties $A \to B$ and $B \to C$ but not $A \to C$, and let the system be in state s_i with *class*$_i$(a) = A. It then executes the sequence *transfer*(:=,a,b)$\cdot$*delete*(a), which leaves it in state s_j with *class*$_j$(b)=B. Clearly execution of this sequence is secure. However, if the operation *transfer*(:=,b,c) is now executed, leaving the system in state s_k with *class*$_k$(c) = C, then the complete sequence *transfer*(:=,a,b)$\cdot$*delete*(a)$\cdot$*transfer*(:=,b,c) is not secure, since information in A has been transferred to C although $A \not\to C$. We observe that this problem would not have arisen if the flow relation $\to$ were transitive (i.e., $A \to B$ and $B \to C \Rightarrow A \to C$). Indeed, if we consider only those systems in which the flow relation is transitive, we can ignore the problem of verifying operation sequences of length m > 1, as the following theorem establishes.

Theorem 2.2-1. Let $\to$ be a transitive flow relation and τ' and τ'' operation sequences. If τ' and τ'' are secure sequences, then $\tau = \tau' \cdot \tau''$ is also a secure sequence.

proof. Consider an arbitrary initial state s such that τ is feasible. If there is none, then τ is trivially secure, so assume such an s exists. Then τ' secure implies that for any $\alpha \in A$ and $\beta \in B$, if $\alpha \overset{s\cdot\tau}{\Longrightarrow} \beta$, then $A \to B$. Also, τ'' secure implies that for any $\beta \in B$ and $\gamma \in C$, if $\beta \overset{s'\cdot\tau''}{\Longrightarrow} \gamma$ then $B \to C$, where $s' = s \cdot \tau'$. Therefore, for

$\alpha \in A$ and $\gamma \in C$, if $\alpha \overset{s\cdot\tau}{\Rightarrow} \gamma$, then A → C by transitivity. Hence, $\tau = \tau'\cdot\tau''$ is secure.

Corollary 2.2-1. Let → be a transitive flow relation. If every operation sequence of length one is secure, then every sequence of operations is secure.

proof. Immediate from preceding theorem.

Given this result, we will first show that the semantics of the problem imply we are justified in assuming that → is transitive. This is done in Section 2.3.1. In the remainder of the thesis, we will then examine protection mechanisms that keep the system in a state such that execution of the next operation is guaranteed secure. Any state that satisfies this property is said to be a safe state. By Corollary 2.2-1, we know that if we can keep the system in a safe state, it will be secure. This result was also established for the MITRE model [BL73a].

Mechanisms for maintaining the system always in safe states are of two types: run-time mechanisms that verify the secure execution of each operation before it is performed (preventing those that are not secure) and certification mechanisms that guarantee that no insecure operation will be attempted. That run-time mechanisms can be used to "leak" information (i.e., violate flow paths - see below) is a compelling argument for certification mechanisms. However, a practical system may need both types of mechanisms to provide the fullest range of defenses.

As an example of using a run-time mechanism to leak information (despite the intention of its designers to prevent leaks!), consider two classes A and B such that $B \not\rightarrow A$, and suppose that process p has access to some information item β in class B, but that it is not allowed to communicate this information to its owner Smith who is in class A (since $B \not\rightarrow A$). However, p may attempt to transfer β from B to A, β times; and though it will fail, the record of the aborted attempts contains the value β! If Smith is permitted to learn of these violations, the information β has effectively been transferred from B to A. This example shows that under certain circumstances, "audit trails" (i.e., records of attempted violations), can be used for improper transfers of information. Rotenberg presents many examples demonstrating that such mechanisms designed to prevent leaks can be used to transmit information in an encoded form in the recorded or observable history of the security mechanism [Ro74]. This form of leakage is one example of a "covert channel".

The problem can be dealt with in systems that use run-time mechanisms. One way simply is to not "sound the alarm" when a security violation is attempted. Although this prevents leakage, it has the disadvantage of not reporting security violations either to the violator (whose intentions may have been innocent, in which case he should be informed of his error) or to the security administrator. The more comforting approach is to sound the alarm and keep an "audit trail" so that all such violations can be scrutinized and the record used as evidence in indictments of wrongdoing. However, for the reasons noted above, access to the audit records must be restricted.

This problem of covert leakage is in principle eliminated with mechanisms that certify that no insecure operations will be attempted.

Certification mechanisms are presented in Chapters 3 and 4. It is important to realize, however, that the need for run-time mechanisms is not obviated by certification mechanisms — they are still necessary, for example, to detect errors wrought by hardware malfunctions that alter the contents of memory locations or errors which may arise from incompletely debugged compilers.

2.3 The Lattice of Information Flow

2.3.1 Derivation

We now examine the mathematical properties of the information flow structure $FS = \langle SC, \rightarrow, \oplus \rangle$ and show that under certain assumptions, FS forms a universally bounded lattice. These assumptions are not arbitrarily chosen, but rather follow from the semantics of information flow. By this we mean either that they are required for consistency or that no generality is lost by them. Consistency means that if one can write a program that effects a permissible flow, then other flows implied by that flow should also be permitted by the flow relation. Stated another way, a program should not be secure by one interpretation and insecure by another. It is important that the flow structure be consistent since it would otherwise be possible to masquerade insecure operations as secure ones.

A *lattice* is any structure consisting of a partially ordered set in which each pair of elements has a well-defined "least upper bound" and "greatest lower bound". A *universally bounded lattice* is a lattice with unique least upper and greatest lower bounds on the entire set. A general discussion of lattices and their properties can be found in

Stone [St73], Birkhoff [Bi67], or Abbott [Ab69]. The reader should note that this is not the first time a lattice has appeared in a study involving the semantics of programs or programming. Scott has shown that lattices, especially continuous functions defined on lattices, apply to data types and control flow diagrams [Sc71]. These structures are partially ordered by a relation based on the idea of "approximation".

To show that FS = $<SC, \rightarrow, \oplus>$ forms a universally bounded lattice, we shall show that for semantic reasons the following four assumptions are justified:

1) $<SC, \rightarrow>$ is a partially ordered set.

2) SC is finite.

3) SC has a lower bound L such that $L \rightarrow A$ for all $A \varepsilon SC$.

4) $\oplus$ is a least upper bound operator on SC; i.e., $A \oplus B$ is the unique least upper bound of A and B for all A, $B \varepsilon SC$.

From these assumptions we shall then establish that there exists a greatest lower bound operator on SC, which we denote by $\otimes$; i.e., $A \otimes B$ is the greatest lower bound of A and B for all A, $B \varepsilon SC$. Therefore, the structure $<SC, \rightarrow, \oplus, \otimes>$ is a lattice with greatest lower bound L and least upper bound, which we denote by H.

To simplify the discussion, we introduce a shorthand underbar notation to denote the $\underline{\text{class}}$ function, where $\underline{a}$ denotes $\underline{\text{class}}(a)$ for object a. Similarly, for an information item α we let $\underline{\alpha}$ denote the class associated with α. For the most part, we do not need to make the state depencency explicit; if we do, we will use the notation $(\underline{a})_i$ to denote $\underline{\text{class}}_i(a)$ (i.e., the class of a in state s_i).

To justify assumption (1), that $<SC,\rightarrow>$ is a partially ordered set, we must show that the relation → should be reflexive, transitive, and anti-symmetric; that is, for all A, B, C, ε SC:

a) $A \rightarrow A$ (reflexive).

b) $A \rightarrow B$ and $B \rightarrow C \Rightarrow A \rightarrow C$ (transitive).

c) $A \rightarrow B$ and $B \rightarrow A \Rightarrow A = B$ (anti-symmetric).

Reflexivity follows from the observation that performing an operation t in state s that does not affect an object a does not alter $\underline{a}$, implying a flow $\underline{a} \rightarrow \underline{a}$ from state s to state $s \cdot t$. Therefore, an inconsistency arises if $A \not\rightarrow A$ for all $A \in SC$. Transitivity follows from the observation that $A \rightarrow B$ implies that it is permissible to move an information item α from an object in A to one in B. Since this puts α in B, $B \rightarrow C$ implies it is then permissible to move α to an object in C and thereby effect a flow from A to C. Therefore, an inconsistency arises if $A \not\rightarrow C$. Anti-symmetry follows from the observation that $A \rightarrow B$ and $B \rightarrow A$ implies anything in one class can be moved into the other. Therefore, one of the two classes is redundant.

To justify assumption (2), that the set of security classes SC is finite, we observe simply that no practical system is capable of supporting an infinite number of security classes.

To justify assumption (3), that there exists a lower bound on SC, we argue that all integers, reals, Booleans, and other data items that normally appear as "constants" in programs should be permitted to flow into all classes (or at least all classes that are bound to the same type of data). Therefore, we assume these values are bound to a lowest security class, which we denote by L. L could also be associated with unoccupied storage locations (i.e., execution of the operation clear(u) could have the effect of setting tag(u) := L). Note also that no generality is lost by the requirement for a single lowest class: if more than one class is desired for different types of constants, no objects need be assigned to the lowest class L.

Before justifying assumption (4), it is necessary to discuss the assumption that $A \oplus B$ is unique — that is, for $\underline{a} = A$ and $\underline{b} = B$, $\underline{f(a,b)} = A \oplus B$ regardless of the function f. It is possible to define different flow relations for different functions; Figure 2.3-1 suggests an example where the function f places its results in class C and function g in class D. One problem with this is the possible semantic ambiguity that arises for results in the range of both f and g — that is, a value $\alpha = f(a,b) = g(a,b)$ for $\underline{a} = A$ and $\underline{b} = B$. Such values are produced from the same operands, and it is possible, under at least some interpretations of the security problem, to say that α has been associated with the wrong class. At this point, we prefer to avoid this murky issue by stipulating that $\oplus$ is independent of the function f used to combine operands. In Section 2.4.1 we show that the effect of a function dependent operator $\oplus$ can be achieved with processes.

To justify assumption (4), that $\oplus$ is a least upper bound operator, we must show that for all A, B, C ε SC:

a) $A \rightarrow A \oplus B$ and $B \rightarrow A \oplus B$.

b) $A \rightarrow C$ and $B \rightarrow C \Rightarrow A \oplus B \rightarrow C$.

Property (a) must hold, since it is semantically absurd to suppose that operands cannot flow into the class of a result generated from them. Also, inconsistencies arise if an operation such as "c := a + b" is

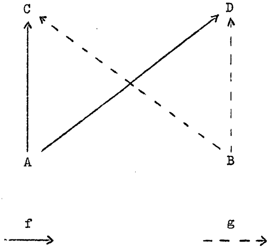


Figure 2.3-1. Example of a Function Dependent Flow Relation.

permitted whereas "c := a" is not, since the latter operation can be performed by executing the former with b = 0. For part (b), consider four objects a, b, c, and d such that $\underline{a} \rightarrow \underline{c}$, $\underline{b} \rightarrow \underline{d}$, and $\underline{c} = \underline{d}$; and consider the following program and corresponding operation sequence:

```
c := a;           transfer(:=,a,c)
d := b;           transfer(:=,b,d)
d := f(c,d)       transfer(f,c,d,d)
```

Execution of this program assigns to d information derived from a and b. Therefore, the flow $\underline{a} \oplus \underline{b} \rightarrow \underline{d}$ is implied semantically, and for consistency we require that the flow relation reflect this fact. Thus for any two classes A and B, $A \oplus B$ is the least upper bound, also referred to as the "join", of A and B. Note that $A \oplus B \rightarrow C$ also implies $A \rightarrow C$ and $B \rightarrow C$ by transitivity and part (a).

A useful property of a least upper bound operator on a partially ordered set is that its domain can be extended to subsets containing more than two elements. This is done as follows: For a non-empty finite subset $\{A_1,\ldots,A_n\}$ of SC, define $\bigoplus$ by

$$\bigoplus\{A_1,\ldots,A_n\} = \begin{cases} A_1 & \text{if } n = 1 \\ \left[\bigoplus\{A_1,\ldots,A_{n-1}\}\right] \oplus A_n & \text{if } n > 1 \end{cases}$$

The following propositions are easily proved by induction on n, using the definition of $\oplus$ for two classes and assumption (4).

Proposition 2.3-1. $\bigoplus$ as defined above specifies the unique least upper bound of $\{A_1,\ldots,A_n\}$.

Since the ordering of elements in the set $\{A_1,\ldots,A_n\}$ is immaterial — i.e., $A_1 \oplus A_2 = A_2 \oplus A_1$ for all A_1, A_2 — the order in which

$\bigoplus\{A_1,\ldots,A_n\}$ is evaluated is immaterial. Therefore, we may write simply

$$A_1 \oplus \ldots \oplus A_n \text{ for } \bigoplus\{A_1,\ldots,A_n\}.$$

Proposition 2.3-2. $A_i \rightarrow B \; (1 \leq i \leq n)$ iff $A_1 \oplus \ldots \oplus A_n \rightarrow B$.

This proposition states that information in objects $a_1,\ldots,a_n$ can flow into an object b if and only if information derived from $a_1,\ldots,a_n$ (e.g., $f(a_1,\ldots,a_n)$) can flow into b (since $\underline{a}_i \rightarrow \underline{b} \; (1 \leq i \leq n)$ iff $\underline{a}_1 \oplus \ldots \oplus \underline{a}_n \rightarrow \underline{b}$). Therefore, from an initial state s_i, the safeness of state $s_j = s_i \cdot \underline{\text{transfer}}(f,a_1,\ldots,a_n,b)$ is established by verifying either

$$\begin{cases} (\underline{a}_k)_i \rightarrow (\underline{b})_j \quad (1 \leq k \leq n) \\ \text{and } (\underline{b})_i \rightarrow (\underline{b})_j \text{ if the contents of b are modified,} \end{cases}$$

or

$$\begin{cases} (\underline{a}_1)_i \oplus \ldots \oplus (\underline{a}_n)_i \rightarrow (\underline{b})_j \text{ if the contents of b are replaced} \\ (\underline{a}_1)_i \oplus \ldots \oplus (\underline{a}_n)_i \oplus (\underline{b})_i \rightarrow (\underline{b})_j \text{ if the contents of b are modified.} \end{cases}$$

Proposition 2.3-2 if the basis of the verification procedures presented in Chapters 3 and 4.

We now prove that assumptions (1) - (4) imply the existence of a greatest lower bound operator on the set of security classes SC. For any pair of classes A and B, define

$$LB(A,B) = \{C \mid C \rightarrow A \text{ and } C \rightarrow B\}.$$

LB(A,B) is the set of possible lower bounds for both classes. Observe that the lowest class L is a member of LB(A,B) by assumption (3), so that LB(A,B) is non-empty for all A, B. Define the operator $\otimes$ to be the least upper bound of the lower bounds:

$$A \otimes B = \bigoplus LB(A,B).$$

Our claim that $\otimes$ is a greatest lower bound operator is established by the following proposition.

Proposition 2.3-3. For a flow structure $FS = \langle SC, \rightarrow, \oplus \rangle$, the operator $\otimes$ is a greatest lower bound operator on SC; i.e., $\otimes$ satisfies the following properties for all A, B, C ϵ SC:

a) $A \otimes B \rightarrow A$ and $A \otimes B \rightarrow B$.

b) $C \rightarrow A$ and $C \rightarrow B \Rightarrow C \rightarrow A \otimes B$.

proof. By Proposition 2.3-1 and $L \; \epsilon \; LB(A,B)$, $\otimes$ is well-defined. By Proposition 2.3-2, $\bigoplus LB(A,B) \rightarrow A$ and $\bigoplus LB(A,B) \rightarrow B$, establishing property (a). If $C \rightarrow A$ and $C \rightarrow B$, then $C \; \epsilon \; LB(A,B)$ and thus $C \rightarrow \bigoplus LB(A,B)$ by Proposition 2.3-1. Therefore property (b) also holds.

Thus $\otimes$ is shown to be the greatest lower bound operator, also called the "meet" operator, of the set of security classes SC. Note that its existence follows from the earlier assumptions and requires no new semantic assumptions. Also note that if $C \rightarrow A \otimes B$, then $C \rightarrow A$ and $C \rightarrow B$ by transitivity and property (b).

As with the least upper bound operator, the greatest lower bound operator $\otimes$ can also be extended to operate on subsets of the security classes SC. In this case for a non-empty finite subset $\{A_1, \ldots, A_n\}$ of SC, we define $\bigotimes$ by

$$\bigotimes \{A_1, \ldots, A_n\} = \begin{cases} A_1 & \text{if } n = 1 \\ [\bigotimes \{A_1, \ldots, A_{n-1}\}] \otimes A_n & \text{if } n > 1 \end{cases}$$

Using inductive arguments, it is easily shown that the following is true.

Proposition 2.3-4. $\bigotimes$ as defined above specifies the unique greatest lower bound of $\{A_1,\ldots,A_n\}$.

Therefore, we can write

$A_1 \otimes \ldots \otimes A_n$ for $\bigotimes\{A_1,\ldots,A_n\}$.

Proposition 2.3-5. $B \to A_i$ $(1 \leq i \leq n)$ iff $B \to A_1 \otimes \ldots \otimes A_n$.

This proposition allows us to verify the security of a set of transfer operations {transfer(f,a,b_1),...,transfer(f,a,b_n)} by showing that $\underline{a} \to \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$. As with Proposition 2.3-2, this result forms the basis of the verification algorithms.

We now observe that the lowest class L is just the greatest lower bound of the entire (finite) set of security classes SC (i.e., $L = \bigotimes SC$). Furthermore, there exists a highest class, denoted by H, which is the least upper bound of the entire set SC (i.e., $H = \bigoplus SC$). Although information in H can be derived from information in any other class, it is not allowed to flow outside of H to another class. As with the requirement that there exists a lowest class L, no generality is lost by the requirement that H exist, since no objects need be assigned to it (this is also true of any class needed to complete the lattice structure).

That the information flow structure FS forms a lattice $\langle SC, \to, \oplus, \otimes\rangle$ with universal bounds L and H now follows from assumptions (1) - (4) and Proposition 2.3-3.

2.3.2 Examples

An example of a structure satisfying the lattice property is derived from a linear ordering on a set of security classes SC:

a) $SC = \{A_1, \ldots, A_n\}$

b) $A_i \rightarrow A_j$ iff $i \leq j$

c) $A_i \oplus A_j \equiv A_{\max(i,j)}$

d) $A_i \otimes A_j \equiv A_{\min(i,j)}$

e) $L = A_1$; $H = A_n$

This is just a generalization of the example illustrated in Figure 2.1-1 (a) for the special case n = 4. This structure is suitable for any system in which the classes are linearly (or hierarchically) ordered. A common case is a government or military system in which the security classes are determined solely from the four security levels: unclassified, classified, secret, and top secret. Another case is found in a system that needs only two classes: unconfidential (L) and confidential (H), with the single security requirement that confidential information cannot flow into an unconfidential object. This case is considered by Denning, Denning, and Graham [DD74] and also by Fenton (using the names <u>null</u> and <u>priv</u> for the lowest class L and the highest class H, respectively) [Fe74a].

A richer structure satisfying the lattice property is derived from a non-linear partial ordering on the set of all subsets that can be constructed from a finite set X:

a) SC = <u>powerset</u>(X)

b) $A \rightarrow B$ iff $A \subseteq B$

c) $A \oplus B \equiv A \cup B$

d) $A \otimes B \equiv A \cap B$

e) $L = \emptyset$; $H = X$

Figure 2.3-2 illustrates this "lattice of subsets" for $X = \{x,y,z\}$. This structure is suitable for any system in which the classes are determined by a set of properties X, and an information item α is not permitted to flow into an object b unless b has at least all of the properties that α has. Consider, for instance, a system that contains medical, financial, and criminal records on individuals (i.e., $X = \{\text{med},\text{fin},\text{crim}\}$). Then medical information would be permitted to flow into only those objects a for which med ε $\underline{a}$, or a combination of medical and financial information would be permitted to flow into only those objects a for which med ε $\underline{a}$ and fin ε $\underline{a}$.

The information flow structure FS of ADEPT [We69] forms a lattice $<SC, \rightarrow, \oplus, \otimes>$ determined by the (cartesian) product ordering of two lattices FS_1 and FS_2, where FS_1 is a linear ordered lattice and FS_2 is a subset lattice. The lattice FS_1 is derived from a linear ordering of a set of Authority Levels $SC_1 = \{A_1,\ldots,A_m\}$ corresponding to the unclassified, confidential, etc. levels of government and military security. FS_2 is derived from an ordering by subsets of the set of all subsets SC_2 determined by a collection of Categories (properties) $X = \{x_1,\ldots,x_n\}$, corresponding to special control compartments used to restrict access by project and area. The information structure FS then forms the lattice $<SC, \rightarrow, \oplus, \otimes>$ defined by:

a) $SC = SC_1 \times SC_2$

b) $<A_i,B> \rightarrow <A_j,B'>$ iff $i \leq j$ and $B \subseteq B'$

c) $<A_i,B> \oplus <A_j,B'> \equiv <A_{\max(i,j)}, B \cup B'>$

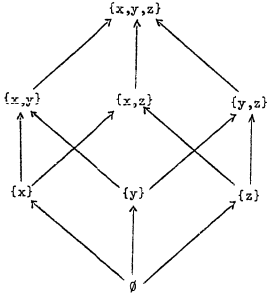


Figure 2.3-2. Lattice of Subsets of X = {x,y,z}.

d) $<A_i, B> \otimes <A_j, B'> \equiv <A_{\min(i,j)}, B \cap B'>$

d) $L = <A_1, \emptyset>$; $H = <A_m, X>$

where B and B' are subsets of X (i.e., elements of SC_2). The Franchise component of ADEPT's security profiles (corresponding to "need to know") is not considered to be a component of its information flow structure, since it is not partially ordered (transitivity is violated). However, this component can be accounted for with a mechanism controlling processes' read and write access rights.

2.3.3 Additional Lattice Properties

We now examine some additional well known properties of lattices and show that they are consistent, as they apply to $\oplus$, with the semantics of the problem:

1) $A \oplus A = A$; $A \otimes A = A$ (indempotent)

2) $A \oplus B = B \oplus A$; $A \otimes B = B \otimes A$ (commutativity)

3) $A \oplus (B \oplus C) = (A \oplus B) \oplus C$;

$A \otimes (B \otimes C) = (A \otimes B) \otimes C$ (associativity)

4) $A \rightarrow B \Rightarrow A \oplus B = B$ and $A \otimes B = A$ (absorption)

The indempotent law is obviously consistent with the semantics since information derived only from information in a class A should remain in class A. The commutativity and associativity laws are consistent since the class of an information item α derived from three objects a, b, and c should be independent of the order in which the information in a, b, and c is combined. For example, if α is derived from the expression "(a + b) + c" and α' from the expression "b + (c + a)", then it should be the case that both $\alpha = \alpha'$ and $\underline{\alpha} = \underline{\alpha}'$. Finally the absorption law is consistent since, if it is permissible to cause a flow from A to B, then information in B in fact represents the combination of information derived from A and B. This law implies that the lowest class L is an identity element over ⊕, so that information combined with data in the lowest security class (e.g., constants) remains in its original class, and that the highest class H is an identity element over ⊗. However L is an absorbing element over ⊗ since L ⊗ A = L, and H is an absorbing element over ⊕ since A ⊕ H = H. (The last case states that once information is linked with highest security data, it remains in the highest security class.)

2.4 Classification of Systems

Mechanisms for maintaining a system in secure states can be classified along two independent, structural dimensions. The first is the number of security classes permitted in each process's working store; sincle-class working stores are an interesting special case. The second is the extent to which objects are permanently bound to security classes, and ranges from "static binding" (no changes permitted) at one extreme to "dynamic binding" (an object changes class according to the class of information affecting its content) at the other.

The first dimension we consider is the structure imposed on the working store (also referred to as the computational memory or virtual memory) W of a system. The *working store* is the subset of all logical objects N in which standard arithmetic and logical operations (e.g., addition and comparison) can be performed. It is distinguished from the *file store* F in that information in a file must be transferred explicitly into the working store of some process (via an *input* instruction) before it can be operated upon or moved to another file.

We assume that $N = W \cup F$. Letting W_p denote the working store of process p (i.e., $W_p \subseteq W$, and p read a or p write a for all $a \in W_p$), we consider two cases: 1) $|W_p| = 1$ (single-class) and 2) $|W_p| \geq 1$ (multi-class).

The single-class case specifies the working store of a process to be a single object, and therefore bound to a single class. Hence, all operations that cause information flow within the working store of a process are necessarily secure, and this includes most of the non-I/O instructions found in programming languages. This case corresponds to the specifications of ADEPT [We69], Rotenberg's Privacy Restriction Monitor [Ro74], the CASE system [Wa74], and the MITRE model [BL73a].

The multi-class case specifies that the working store of a process may be more than one object, and therefore bound to more than one class. Here, all operations that cause flow within the working store of a process are not necessarily secure, so mechanisms are needed to guarantee the secure execution of all non-I/O as well as I/O instructions. This case corresponds to the abstract model proposed by Fenton [Fe73, Fe74a,Fe74b] and the systems described in Chapters 3 and 4 of this thesis.

The alert reader may wonder why the single-class case restricts W_p to a single object, rather than a set of objects from the same security class. The reason is that it is impossible to combine the notion of dynamic binding with a single-class, multiple object, working store, since the classes of objects bound dynamically can change over time.

The single-class model is useful for studying systems in which a process's working store is the only private store it has and files are generally subject to access by many processes. In contrast, systems which permit objects in virtual memory to be shared (i.e., the working store of a process is not completely private) must usually be treated with the multi-class model.

In the context of single-class systems, the only *transfer* operations that require verification are I/O operations that transfer data either from a file a into the working store W_p of a process p (i.e., *transfer*(*input*,a,W_p)) or from a working store W_p into a file b (i.e., *transfer*(*output*,W_p,b)). Therefore, the security requirements will be satisfied if these operations satisfy the conditions:

i) $\underline{a} \oplus \underline{W}_p \rightarrow \underline{W}_p$, for *transfer*(*input*,a,W_p);

ii) $\underline{b} \oplus \underline{W}_p \rightarrow \underline{b}$, for *transfer*(*output*,W_p,b).

In the context of multi-class systems, I/O operations such as the above need verification as well as all operations that cause a flow of information within a process's working store (e.g., assignment operations).

The second dimension of security system classification is the method of associating (binding) objects to security classes. *Static binding* means that the security class of an object is constant or invariant over all states of the system unless explicitly altered by the *reclassify* operation. This is the type of binding used in the CASE system, the MITRE model, and in Fenton's model. *Dynamic binding* means that the security class of an object varies with its contents as follows: if an operation *transfer*(f,a_1,...,a_n,b) is performed, the security requirements are satisfied if the class of b is updated according to the assignments:

$\underline{b} := \underline{a}_1 \oplus \cdots \oplus \underline{a}_n$, if the contents of b are replaced; or

$\underline{b} := \underline{a}_1 \oplus \cdots \oplus \underline{a}_n \oplus \underline{b}$, if the contents of b are modified.

Consequently, execution of a *transfer* operation in a dynamic binding system is *always secure* (provided, of course, the updating mechanism functions correctly). In the case of dynamic binding, security classes are actually associated with the information stored in an object rather than with the object itself.

A system based purely on dynamic binding may not be practical: most users, along with their private files, are usually considered to belong to fixed classes. Hence, we must also consider realistic combinations of static and dynamic binding — that is, some objects are statically bound and others are dynamically bound. Such are the specifications of ADEPT and Rotenberg's Privacy Restriction Monitor.

Recall from the results of Section 2.2 that security is guaranteed if the system can be kept in a safe state; i.e., execution of the next operation cannot cause a flow from a class A to a class B unless $A \rightarrow B$. Recall also that the only operations that can potentially cause a flow from A to B $(A \neq B)$ are *create object*, *reclassify*, and *transfer*. Now, the security of object creation is assured if the system always clears memory before assigning it to a new object. The security of reclassification is assured by a simple mechanism that checks $\underline{a} \rightarrow A$ whenever an operation *reclassify*(a,A) is attempted to change the class of object a. The security of *transfer* operations is, therefore, the only remaining problem to consider when analyzing the security requirements of the different types of systems. This will be done in the following sections which describe five types of systems.

2.4.1 Single-Class, Static Binding

To guarantee the secure execution of all transfer operations in a single-class system with static binding of objects to security classes, it is necessary and sufficient to show that if a process p executes an instruction "input x from a" then $\underline{a} \oplus \underline{W_p} \rightarrow \underline{W_p}$ and if it executes an instruction "output x to b" then $\underline{b} \oplus \underline{W_p} \rightarrow \underline{b}$, where $\underline{W_p}$ is p's working store. Also, by the results of Proposition 2.3-2 and reflexivity, these conditions respectively reduce to $\underline{a} \rightarrow \underline{W_p}$ and $\underline{W_p} \rightarrow \underline{b}$. Since execution of these instructions is conditioned on process p having the necessary read or write access privileges, security is guaranteed if in every state s,

a) p read a => $\underline{a} \rightarrow \underline{W_p}$, and

b) p write b => $\underline{W_p} \rightarrow \underline{b}$.

Since grant is the only operation that adds new read and write relations and reclassify is the only operation that changes the class of an object, a safe state can be maintained with a simple mechanism attached to these operations that verifies conditions (a) and (b) above. Hence, in this case, it is not necessary to monitor each I/O operation.

If a process p has an associated clearance (see Section 2.2), it could be used to determine the class of $\underline{W_p}$; that is, if a process p is created, its working store W_p would be created and assigned to p by execution of the operation sequence create(W_p,$\underline{p}$)•grant(p,W_p,rw), where $\underline{p}$ denotes clearance(p). In this case, $\underline{p}$ specifies the highest security class that p can read from, and the lowest security class that it can write into. If $a_1,\ldots,a_m$ are the files p can read from, and $b_1,\ldots,b_n$ the files p can write into, then security requires that

$$\underline{a}_1 \oplus \dots \oplus \underline{a}_m \rightarrow \underline{p} \rightarrow \underline{b}_1 \otimes \dots \otimes \underline{b}_n$$

by conditions (a) and (b) above and the results of Propositions 2.3-2 and 2.3-5. See Figure 2.4-1. For example, in a government or military system, a process having a "secret" clearance would not be able to read from a "top secret" file or write into an "unclassified" one (even if it had processed only unclassified information during execution). Such are the CASE system and the MITRE model. In fact, it does not make sense to design a single-class, static binding system without such a clearance function, since some mechanism is needed to determine the class of a process's computational store when it is created.

In certain cases the effect of a function dependent $\oplus$ (i.e., $\underline{a} \oplus \underline{b}$ depends on the operation performed on $\underline{a}$ and $\underline{b}$ — see Section 2.3) can be achieved with processes. Let p and q be processes such that

a) p and q read from common input files a and b, where $\underline{a} \oplus \underline{b} \rightarrow \underline{p}$ and $\underline{a} \oplus \underline{b} \rightarrow \underline{q}$;

b) p writes into output file c, where $\underline{p} \rightarrow \underline{c}$;

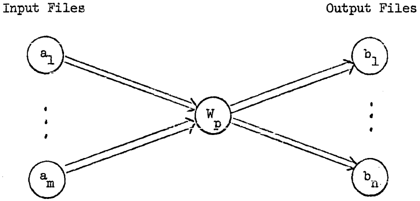
c) q writes into output file d, where $\underline{q} \rightarrow \underline{d}$; and

d) $\underline{c} \neq \underline{d}$.

Clearly, all security requirements are satisfied. Now, execution of p has the effect of combining information in classes $\underline{a}$ and $\underline{b}$ and putting the result in class $\underline{c}$; execution of q has the effect of combining information in classes $\underline{a}$ and $\underline{b}$ and putting the result in class $\underline{d}$. Therefore, p implements $\underline{a} \oplus \underline{b} = \underline{c}$, and q implements $\underline{a} \oplus \underline{b} = \underline{d}$.

2.4.2 Single-Class, Dynamic Binding

To guarantee the secure execution of all transfer operations in a single-class system with dynamic binding of objects to security



$$\underline{a}_1 \oplus \ldots \oplus \underline{a}_m \rightarrow \underline{p} \rightarrow \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$$

Figure 2.4-1. Information Flow in a Single-Class, Static Binding System.

classes, it is necessary and sufficient to attach to the I/O processor a mechanism that correctly updates the class of W_p whenever a process p executes an instruction "input x from a" and the class of a file b whenever p executes an instruction "output x to b", namely

$\underline{W}_p := \underline{W}_p \oplus \underline{a}$, for input, and

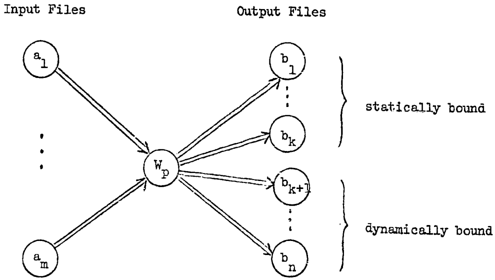
$\underline{b} := \underline{W}_p \oplus \underline{b}$, for output.

Such is the mechanism used by Rotenberg's Privacy Restriction Monitor for controlling flow to segments and the mechanism used by ADEPT for changing the "history" class associated with the working store of a process.

The existence of a clearance function does not seem necessary in this type of system since the automatic class updating mechanism attached to the I/O processor makes it unnecessary to constrain what files a process can access — security is always guaranteed.

2.4.3 Single-Class, Static and Dynamic Binding

The concepts of static and dynamic binding can be combined in a single system, with some objects statically bound and others dynamically bound. For example, in ADEPT, all existing files are statically bound, while new files and the working stores of processes are dynamically bound. One possible specification for a secure system incorporating both types of binding and making use of a clearance function is now outlined. (See Figure 2.4-2). Each process p is initialized with a dynamically bound working store W_p in the lowest security class L; as it inputs information from files, $\underline{W}_p$ is updated as described for the dynamic binding case. However, unlike a pure dynamic binding system, p is permitted to gain (and keep) read access to a file a only if the



$$\underline{a}_1 \oplus \dots \oplus \underline{a}_n \to \underline{p}$$

$$\underline{W}_p \to \underline{p}$$

$$\underline{W}_p \to \underline{b}_1 \otimes \dots \otimes \underline{b}_k$$

Figure 2.4-2. Information Flow in a Single-Class System with Static and Dynamic Binding.

current class of file a satisfies the constraint $\underline{a} \rightarrow \underline{p}$. It is free to acquire write access to a file b (except as constrained by determinacy requirements, "need to know", or other conditions), though it is not permitted to exercise such a right if b is statically bound and the current class of $\underline{W_p}$ does not satisfy the constraint $\underline{W_p} \rightarrow \underline{b}$ (if b is dynamically bound, it can, of course, exercise the right since $\underline{b}$ will be updated to $\underline{b} := \underline{b} \oplus \underline{W_p}$). It is precisely in this last requirement that ADEPT has (from our point of view) a security flaw. In ADEPT, a process with "top secret" clearance can write into any statically bound file b for which $\underline{b} \rightarrow \underline{p}$ (rather than $\underline{W_p} \rightarrow \underline{b}$), making it permissible to write "top secret" data into an "unclassified" file!

In such a system, we define a <u>routing process</u> to be any process p with highest clearance (i.e., $\underline{p} = H$), since such a process is free (insofar as the model is concerned) to read any file and thus route any information in a class A to a file in a resulting class B, provided only that A → B. Routing processes are particularly applicable to data base systems in which the data is distributed among several files associated with different classes, is to be accessed directly only by a specified set of data manipulation processes (the routing processes), and is to be transmitted to users according to a specified flow relation.

There are two major limitations with the single-class systems outlined above. The first is that it is not possible to distinguish different classes of data in a process's working store, since everything must get lumped into a single class. Hence, a process cannot directly access and manipulate data in segments (or registers) belonging to different classes as would be required by certain data base systems where it is not practical to physically isolate the different classes of information into separate files. The second is that it is not possible to account for the shared use of segments (or registers) between two processes p and p'. Hence, it is not possible to model inter-process communication via message buffers, semaphores, and so on.

2.4.4 Multi-Class, Static Binding

The primary difficulty with guaranteeing security in a multi-class system lies in detecting (and monitoring) all transfer operations. This is because all transfer operations in a program are not explicitly specified — or indeed even executed! We showed in the previous sections that the only flows possible in a single-class system are "explicit" ones caused directly by I/O operations. When we turn to multi-class systems we find that some transfer operations applied within the working store cause "implicit" flows in addition to the explicit flows for which they were programmed; this "extra" flow complicates security checking. As an example, consider the statement "if a = 0 then b := 0"; if $b \neq 0$ initially, testing b = 0 on termination of this statement is tantamount to knowing whether a = 0 or not. In other words, information flows from a to b regardless of whether or not the then clause is executed.

To deal with this problem, we distinguish between two types of flow: "explicit" and "implicit". Explicit flow to an object b occurs as the result of executing any statement (e.g., assignment or I/O) that directly transfers to b information derived from some sequence of operand objects $a_1,\ldots,a_n$. Examples of explicit flow are:

Statement	Explicit Transfer
b := a	transfer(:=,a,b)
b := a * b	transfer(*,a,b,b)
input b from a	transfer(input,a,b)
output a to b	transfer(output,a,b)

Implicit flow to b occurs as the result of executing or not executing a statement that causes an explicit flow to b when that statement is conditioned on the value of a Boolean expression. Examples of statements that cause implicit as well as explicit flow are:

Statement	Explicit Transfer	Implicit Transfer
if a = 0 then b := 0	transfer(:=,0,b)	transfer(=,a,0,b)
if a = 0 then input b from c	transfer(input,c,b)	transfer(=,a,0,b)

while a > 0 do
b := b + 1;
a := a - 1
end
	transfer(+,b,1,b)
	transfer(-,a,1,a)
		transfer(>,a,0,b)
		transfer(>,a,0,a)

if a = 0 then if b = 0
then c := d
	transfer(:=,c,d)	transfer(=,a,0,c)
		&
		transfer(=,b,0,c)

To guarantee the secure execution of a program in the presence of implicit flows, it is convenient to consider an abstract representation of programs that preserves the flows but not necessarily all of the original structure. Define recursively an abstract program Φ by:

1. Φ is an elementary statement; that is, an assignment or I/O statement.
2. There exist Φ_1 and Φ_2 such that $\Phi = \Phi_1;\Phi_2$.
3. There exist $\Phi_1,\ldots,\Phi_m$ and an m-valued expression ξ such that $\Phi = \xi:\Phi_1,\ldots;\Phi_m$.

Step (1) declares simple statements as abstract programs. Step (2) declares sequences of simpler programs as abstract programs. Step (3) declares conditional structures, in which an expression ξ selects among alternative programs, as abstract programs.

The conditional structure is used here to represent all conditional (including iterative) statements found in programming languages. For example, "if ξ then Φ_1 else Φ_2" is represented by "$\xi:\Phi_1,\Phi_2$". Both "if ξ then Φ_1" and "while ξ do Φ_1" are represented by "$\xi:\Phi_1$", and "do case ξ of $\Phi_1;\ldots;\Phi_m$" is represented by "$\xi:\Phi_1,\ldots,\Phi_m$". Structures arising from the use of goto statements can also be represented by the conditional structure, but to do so is more difficult (see Section 3.3).

The security requirements for any program of the above form are now stated simply. Firstly, for any elementary statement E, $\Phi = E$ is secure if any explicit flow caused by E is secure (no implicit flow is possible). Specifically, if E represents an operation transfer$(f,a_1,\ldots,a_n,b)$, the security requirements stipulate that

$\underline{a}_1 \oplus \ldots \oplus \underline{a}_n \rightarrow \underline{b}$, if E replaces the contents of b;

$\underline{a}_1 \oplus \ldots \oplus \underline{a}_n \oplus \underline{b} \rightarrow \underline{b}$, if E modifies the contents of b.

For example, if the unit of protection is a single register and E is the statement " d := a + b * c", security requires that $\underline{a} \oplus \underline{b} \oplus \underline{c} \rightarrow \underline{d}$. Secondly, $\Phi = \Phi_1;\Phi_2$ is secure if both Φ_1 and Φ_2 alone are secure (see Theorem 2.2-1). Thirdly, $\Phi = \xi:\Phi_1,\ldots,\Phi_m$ is secure if each Φ_k alone is secure and all implicit flows from ξ are secure. To make this specific, let $V(\Phi) = \{b_1,\ldots,b_n\}$ denote the set of objects that information can explicitly flow to in Φ – that is, for each b_i $(1 \le i \le n)$, there is an operation transfer$(\ldots,b_i)$ is some Φ_k; all implicit flow is secure if $\underline{\xi} \rightarrow \underline{b}_i$ $(1 \leq i \leq n)$ or equivalently (by Proposition 2.3-5) if $\underline{\xi} \rightarrow \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$ after any execution of the structure.

The foregoing requirements on multi-class systems with static binding can be implemented as run-time checks (e.g., on tags associated with objects in the working store) or as compiler-verification procedures (to be discussed in Chapter 3).

An interesting example of a multi-class, static-binding system has been proposed by Fenton [Fe73,Fe74a]. Fenton chose to cast his results in the context of a Minsky machine [Mi67], which is an automaton modeling a multi-register computer. Fenton extends the Minsky machine by permanently tagging each register with the class to which information residing in it belongs; this corresponds to the tag function defined earlier. The Minsky machine has only two instructions in addition to a halt instruction: the "increment" instruction for any register a, having the effect "a := a + 1"; and the "decrement and jump if zero" instruction, having the effect "if a = 0 then goto m else a := a - 1". (Minsky has shown that this machine has universal computing power. Fenton used it as a formal model within which to prove a secure environment.) Now, the decrement instruction permits information to flow (implicitly) from a to any register affected by any statement reachable from the statement labelled m. To enable flow-checking, Fenton tags the program counter p, updating the class of p according to $\underline{p} := \underline{p} \oplus \underline{a}$ on execution of the above decrement instruction (in Fenton's model, $\underline{p}$ is the only class that can be changed). The successful execution of any increment or decrement instruction applied to register a depends on the pre-condition $\underline{p} \oplus \underline{a} \rightarrow \underline{a}$, which is verified prior to executing the instruction. It is the nature of the above mechanism that the sequence of classes of p is nondecreasing in security. To permit the class of p to be decreased, Fenton introduces a stack of p-classes and a new instruction "<u>if</u> a = 0 <u>then</u> <u>call</u> m <u>else</u> a := a - 1", where the operation "<u>call</u> m" pushes $(p,\underline{p})$ onto the stack and sets $(p,\underline{p}) := (m,\underline{p} \oplus \underline{a})$. An instruction <u>return</u> is also provided so that the block of instructions beginning at location m can return control, pop the stack, and continue execution following the above <u>if</u>-statement. Fenton shows that secure execution is guaranteed with the new instruction added.

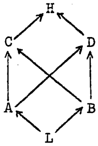
Because he develops his $\oplus$ operator to be consistent with the properties of the Minsky machine without considering the semantics of the information flow problem, Fenton's flow structure $<SC, \rightarrow, \oplus>$ is not a lattice, though it is a partial ordering of SC. The discrepancy is that his $\oplus$ operator does not satisfy the least upper bound property, "$A \rightarrow C$ and $B \rightarrow C \Rightarrow A \oplus B \rightarrow C$", and is, as a consequence, not associative. The reason for this "semantic defect" can be understood by observing that, in a Minsky machine, it is impossible to extract directly information from two registers and place the result in a third register. It is possible only to use the contents of one register to modify the contents of another. (For example, the only Minsky program to cause the sum a + b to appear in register c, assuming c = 0 initially, uses the circuitous procedure of adding a to b then copying the result to c.) For this reason, Fenton defined his $\oplus$ operator as follows:

$$A \oplus B = \begin{cases} A & \text{if } B \rightarrow A \quad \text{(B can modify A)} \\ B & \text{if } A \rightarrow B \quad \text{(A can modify B)} \\ H & \text{otherwise (neither A nor B can modify the other).} \end{cases}$$

An example of a complete structure satisfying these properties is given in Figure 2.4-3. A requirement such as "the combination of information in classes A and B can be placed in class C" is handled by defining A → C and B → C (i.e., $A \oplus C = C$ and $B \oplus C = C$), so that a program can use information in classes A and B to modify registers of class C and create the desired combination piecemeal in registers of C. Since it is possible to have A → C and B → C but not $A \oplus B \rightarrow C$, $\oplus$ is not a least upper bound operator.

When considering how to modify Fenton's $\oplus$ operator so that it becomes a lattice operator, one quickly discovers that it cannot be done without adding new security classes. Figure 2.4-4 shows a modification of the structure of Figure 2.4-3 to this end, a new class AB being added. Since Fenton's structure can be extended to a lattice, one can inquire whether his structure is semantically meaningful. The answer is, of course, affirmative in the context of the Minsky machine for which it was derived.

The problem under consideration — secure information flow in multi-class systems with static binding of security classes — can be solved completely by a compile-time mechanism, without the need for tagged architecture or run-time hardware checking mechanisms (see Chapter 3). However, the run-time mechanism could be used as a redundant check in case, for example, a hardware malfunction introduced an error into a previously certified program. (See Fabry [Fa73] for a general discussion of redundancy checks to enhance system reliability.) The compile-time mechanism has the further attraction that it eliminates the possibility of "covert leakages" caused by tripping alarms built into run-time mechanisms.

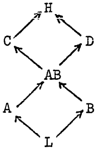


Graph of →

	L	A	B	C	D	H
L	L	A	B	C	D	H
A	A	A	H	C	D	H
B	B	H	B	C	D	H
C	C	C	C	C	H	H
D	D	D	D	H	D	H
H	H	H	H	H	H	H

Matrix of ⊕

Figure 2.4-3. Example of an Information Flow Structure Satisfying Fenton's Model.



Graph of →

	L	A	B	AB	C	D	H
L	L	A	B	AB	C	D	H
A	A	A	H	AB	C	D	H
B	B	H	B	AB	C	D	H
AB	AB	AB	AB	AB	C	D	H
C	C	C	C	C	C	H	H
D	D	D	D	D	H	D	H
H	H	H	H	H	H	H	H

Matrix of ⊕

Figure 2.4-4. Example of Figure 2.4-3 Extended to a Lattice.

2.4.5 Multi-Class, Dynamic Binding

Mechanisms for guaranteeing security when all (or some) objects are bound dynamically to security classes require both compile-time and run-time support and are the most complex of all. As an example, consider a system in which the unit of protection is a single register and all objects are bound dynamically to classes. Suppose the following program (proposed by Fenton [Fe73] is executed to set b equal to the value of a (assumed to be either 0 or 1 initially):

```
b := 0;
c := 0;
if a = 0 then c := 1;
if c = 0 then b := 1
```

The "naive" run-time mechanism (see also [DD74]) for dynamically binding registers to classes operates by updating the class of a register whenever an assignment is made to it. For the above program, this proceeds as follows:

Statement	Action
b := 0	b := L
c := 0	c := L
if a = 0 then c := 1	if c := 1 is executed, c := a
if c = 0 then b := 1	if b := 1 is executed, b := c

When this program terminates, the class of b is the lowest class L (i.e., b = L) irrespective of the class of a, but the value of b is identically that of a. This is because the mechanism does not account for the implicit flow that occurs when the statements "c := 1" and "b := 1" are not executed. Therefore, a security violation results unless a = L.

The above program illustrates that a run-time mechanism is not sufficient to guarantee security since implicit flows that occur in the absence of explicit flows (from assignment or I/O operations) go undetected. For this reason, Fenton dismisses the problem as insoluble in his framework which relies solely on a run-time mechanism. In Chapter 4 we show that the problem can be solved with a compile-time procedure that inserts special code (if need be) into a program so that all implicit flows are accounted for. In the program given above, for example, the compiler would insert code so that execution of the statement

    if a = 0 then c := 1

results in the assignment

    c := a

even if a ≠ 0. The point is, the problem of implicit flow can be solved if one is willing to augment the run-time mechanism with a compile-time mechanism.

2.5 Determinacy Requirements

The existence of appropriate security flow controls does not mean that the system is fully secure if information can still be communicated along "covert channels" [La73]. One such case arises if the system permits the class of a dynamically bound object to be observed, and the final class of the object can be made to depend on the relative speeds of parallel processes; this non-determinacy can be used to transmit information to an observer. For such systems, additional constraints may be required to guarantee determinate operation. Determinacy is, of course, usually considered desirable in its own right, because it guarantees the reproducibility of a system's actions under arbitrary timings of processes that share information.

To formulate conditions that guarantee determinate behavior, we consider first the simple case of a single-class system under dynamic binding of security classes (see Section 2.4.2). We than extend the results to a single-class system under both static and dynamic binding of security classes (see Section 2.4.3) and to a multi-class system under dynamic binding of security classes (see Section 2.4.5). In all cases, we assume that the set of all objects and processes is constant over the lifetime of the system (i.e., no create or delete operations are performed). Our treatment closely follows that of Coffman and Denning [CD73].

2.5.1 Single-Class, Dynamic Binding

We assume that each process p performs a computation in its private working store W_p using as input information from a domain set $D_p \subseteq F$ and producing as output information in a range set $R_p \subseteq F$, where F is the set of files in the system. The domain D_p is the set of files to which p has read access; the range R_p is the set to which it has write access. We make the restriction that D_p and R_p do not change during the execution of p.

The set P of all processes in existence is partially ordered by a relation $<$, where $p < p'$ means that p must terminate before p' can begin. The system $\Pi = \langle P, < \rangle$ is called a process system. Figure 2.5-1 is an example of a process system represented by a precedence graph. Associated with each process p is an "initiation event" corresponding to the operation initiate(p), denoted by "[p". Similarly, there is a "termination event" corresponding to terminate(p), denoted by "p]".

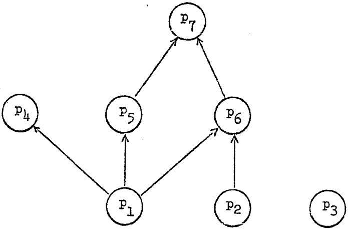


Figure 2.5-1. Example of a Process System.

For the purposes of proving determinacy, no generality is lost by associating all input operations with the initiation of p and all output operations with the termination of p. The reason is that such a proof relies only on the assumption that p is determinate — i.e., that the class assigned to an output file depends uniquely on the classes of p's input files, as long as no input file classes change during p's execution; exactly when the classes of output files are updated is, therefore, irrelevant.

An execution sequence of a process system $\Pi = <P, \prec>$ is a string $\rho = r_1 r_2 \dots r_{2n}$, n being the number of processes in P, and each r_i $(1 \leq i \leq 2n)$ being a process initiation or termination event such that:

1) For each $p \in P$, "[p" and "p]" appears exactly once in ρ,

2) "[p" precedes "p]" in ρ, and

3) "p]" precedes "[p'" in ρ when $p \prec p'$.

A valid execution sequence for the system of Figure 2.5-1 is

$$[p_1\ [p_2\ p_1]\ [p_4\ [p_5\ p_2]\ [p_6\ p_5]\ p_6]\ [p_7\ [p_3\ p_3]\ p_7]\ p_4].$$

More than one valid execution sequence will exist for Π unless $\prec$ is a linear ordering relation.

Let $\sigma(\rho) = s_0 s_1 \dots s_{2n}$ denote the state sequence corresponding to an execution sequence ρ; that is, s_0 is the initial state, and $s_i = s_{i-1} \cdot r_i$ $(1 \leq i \leq 2n)$.

Define a <u>class sequence</u> $c(a,\rho)$ for each file $a \in F$ to be the sequence of classes $c_1 c_2 \dots c_m$ to which a is assigned during the execution sequence ρ. Letting λ denote the empty sequence, we have:

$$c(a,\lambda) \quad = \quad (\underline{a})_0$$

$$c(a,r_1 \ldots r_k) = \begin{cases} (c(a,r_1 \ldots r_{k-1}) \ (\underline{a})_k) & \text{if } r_k = p] \text{ and } a \in R_p \\ c(a,r_1 \ldots r_{k-1}) & \text{otherwise} \end{cases}$$

where $(\underline{a})_i$ denotes the class of a in state s_i. A condition for determinacy is formulated in terms of the <u>final value</u> of a class sequence $c(a,\rho) = c_1 c_2 \ldots c_m$. Letting $\nu(a,\rho)$ denote c_m, a protection system is defined to be <u>determinate</u> if $\nu(a,\rho) = \nu(a,\rho')$ for all files $a \in F$ and all execution sequences ρ and ρ'. An example of a system with two independent processes that is not determinate is illustrated in Figure 2.5-2. Here $\nu(b,\rho) \neq \nu(b,\rho')$ for file b and sequences ρ and ρ'.

We have intentionally defined determinacy in terms of final classes rather than class sequences (as is done analogously in Coffman and Denning). Determinacy with respect to final classes is important for two reasons. First, the final class of a file a written by processes $p_1,\ldots,p_n$ should be uniquely determined by the information $p_1,\ldots,p_n$ write into a and not by the order in which $p_1,\ldots,p_n$ execute. Second, an observer should not be able to obtain unauthorized information by testing the final class of file a (e.g., by attempting a read or write operation on a). Determinacy with respect to class sequences is not important if no observer is allowed to examine the class sequence of file a.

<u>Theorem 2.5-1.</u> A process system $\Pi = \langle P, \prec \rangle$ is determinate if for all p and $p' \in P$ one of the following conditions is true:

a) $p \prec p'$ or $p' \prec p$, or

b) $R_p \cap D_{p'} = D_p \cap R_{p'} = \emptyset$.

$\Pi = \langle \{p_1, p_2\}, < \rangle$

$F = \{a,b,c\}$; $(\underline{a})_0 = (\underline{b})_0 = L$, $(\underline{c})_0 = H$

$D_{p_1} = a$, $R_{p_1} = b$ (i.e., p_1 inputs from a and outputs to b)

$D_{p_2} = c$, $R_{p_2} = b$ (i.e., p_2 inputs from c and outputs to a)

$\rho = [p_1\ p_1]\ [p_2\ p_2]$

$\rho' = [p_2\ p_2]\ [p_1\ p_1]$

Tables showing the classes of the files a, b, and c and of the working stores W_{p_1} and W_{p_2} after each operation is performed for ρ and ρ':

ρ

$[p_1\ p_1]\ [p_2\ p_2]$

a	L	L	L	L	H
b	L	L	L	L	L
c	H	H	H	H	H
W_{p_1}	L	L	L	L	L
W_{p_2}	L	L	L	H	H

ρ'

$[p_2\ p_2]\ [p_1\ p_1]$

a	L	L	H	H	H
b	L	L	L	L	H
c	H	H	H	H	H
W_{p_1}	L	L	L	H	H
W_{p_2}	L	H	H	H	H

Class sequences for b:

$c(b,\rho) = L\ L \Rightarrow \nu(b,\rho) = L$
$c(b,\rho') = L\ H \Rightarrow \nu(b,\rho') = H$

Figure 2.5-2. Example of a Non-Deterministic System.

proof. The proof proceeds by considering a sequence of subsystems $\Pi_1,\ldots,\Pi_n$ in which $\Pi_1 = \langle\{p_1\}, \emptyset\rangle$ for an initial process p_1 of P (i.e., p_1 has no predecessors), $\Pi_n = \Pi$, and $\Pi_{k+1} = \langle P_{k+1}, \prec_{k+1} \rangle$ where P_{k+1} is obtained from P_k by adding any process $p \in P - P_k$ all of whose predecessors (if any) are in P_k and $\prec_{k+1}$ is the restriction of $\prec$ on the subset P_{k+1}. Π_1 is determinate under the assumption that individual processes are determinate. Assume that Π_k is determinate, and let p be the process added to P_k to obtain P_{k+1}. Let $a \in F$ have initial class A and suppose Π_k leaves file a with final class $\underline{a}$. We wish to show that, in Π_{k+1}, the final class of a is $\underline{a}$ if $a \notin R_p$ and $\underline{a} \oplus \underline{W}_p$ if $a \in R_p$, where W_p is p's working store. We must also show that $\underline{a}$ is not affected by the presence of p and that $\underline{W}_p$ is unique. Conditions (a) and (b) both imply that p cannot alter domain classes of any $p' \in P_k$ prior to the initiation of p'; therefore, $\underline{W}_{p'}$ is unique for all $p' \in P_k$. Hence, $\underline{a}$ cannot be altered by the presence of p. Since the classes of D_p are produced by Π_k and since p is determinate, the class $\underline{W}_p$ when p performs its last write operation on a is unique.

Now, for $a \notin R_p$ it is clear that, in Π_{k+1}, the final class of a is $\underline{a}$. For $a \in R_p$ the final class of a, in Π_{k+1}, is

$$\begin{aligned} & A \oplus \left[\bigoplus\{\underline{W}_{p'} \mid p' \in P_{k+1} \text{ and } a \in R_{p'}\}\right] \\ = {} & A \oplus \left[\bigoplus\{\underline{W}_{p'} \mid p' \in P_{k} \text{ and } a \in R_{p'}\}\right] \oplus \underline{W}_p \\ = {} & \underline{a} \oplus \underline{W}_p, \end{aligned}$$

Therefore, the subsystem Π_{k+1} produces unique final classes for every file and is determinate. Repeating the argument for k = 1, 2, ..., n-1 establishes the determinacy of Π.

If one wants to guarantee class sequence determinacy, it is sufficient to add the condition $R_p \cap R_{p'} = \emptyset$ to condition (a) of the theorem statement. In this case, the determinacy proof resembles that given in Coffman and Denning [CD73: Theorem 2.1, p. 39].

2.5.2 Single-Class, Static and Dynamic Binding

Let F^s denote those files statically bound to security classes, and let F^d denote those files dynamically bound to security classes. Also, let P^s denote those processes whose working stores are bound statically, and let P^d denote those processes whose working stores are bound dynamically. We first observe that any file in F^s can be ignored from determinacy considerations, since its class is invariant and therefore completely independent of the behavior of all processes. Second, the interaction between two processes p and p' both in P^s can be ignored since $\underline{W}_p$ and $\underline{W}_{p'}$ are predetermined by clearance(p) and clearance(p') respectively (see Section 2.4.1), and therefore unique for all execution sequences. Hence it is not necessary in this case to account for the conditions $R_p \cap D_{p'} \neq \emptyset$ or $D_p \cap R_{p'} \neq \emptyset$. However, the interaction between a process $p \in P^s$ and $p' \in P^d$ cannot be ignored since $\underline{W}_{p'}$ is determined by $D_{p'}$, and therefore is affected by the behavior of p if $R_p \cap D_{p'} \cap F^d \neq \emptyset$ (although the condition $D_p \cap R_{p'} \neq \emptyset$ cannot affect $\underline{W}_p$). Finally, the interaction between two process p and p' both in P^d cannot be ignored since $R_p \cap D_{p'} \cap F^d \neq \emptyset$ affects $\underline{W}_{p'}$ and $D_p \cap R_{p'} \cap F^d \neq \emptyset$ affects $\underline{W}_p$ as before. These observations lead to the following theorem.

Theorem 2.5-2. A process system $\Pi = \langle P, < \rangle$ is determinate if for all p and $p' \in P$ one of the following conditions is true:

a) $p, p' \in P^s$, or

b) $p < p'$ or $p' < p$, or

c) $R_p \cap D_{p'} \cap F^d = \emptyset$ for $p \in P^s$ and $p' \in P^d$, or

d) $R_p \cap D_{p'} \cap F^d = D_p \cap R_{p'} \cap F^d = \emptyset$ for $p, p' \in P^d$.

2.5.3 Multi-Class, Dynamic Binding

This type of system would appear to be more difficult to analyze since processes may share objects in their working stores (i.e., $W_p \cap W_{p'} \neq \emptyset$ is possible) and since a transfer operation replaces (rather than extends) the contents of the receiving object in the case of a scalar variable, for example. We deal with the first problem by partitioning the set of objects in the working store W into those that are shared, denoted W^{sh}, and those that are private, denoted W^{pr}, and defining each process p to be a computation from a domain $D_p \subseteq F \cup W^{sh}$ to a range $R_p \subseteq F \cup W^{sh}$ using its private working store, $W_p \cap W^{pr}$, for intermediate results. We deal with the second by taking into account the interaction between two processes p and p' when $R_p \cap R_{p'} \cap W^{rep} \neq \emptyset$, where $W^{rep} \subseteq W$ is the set of replacable objects (since the final class of an object $a \in W^{rep}$ may depend on the order in which the processes write into a). The determinacy requirements now follow:

Theorem 2.5-3. A process system $\Pi = \langle P, < \rangle$ is determinate if for all p and $p' \in P$ one of the following conditions is true:

a) $p < p'$ or $p' < p$, or

b) $R_p \cap D_{p'} = D_p \cap R_{p'} = R_p \cap R_{p'} \cap W^{rep} = \emptyset$.

Note that the above requirements are identical to those of the single-class, dynamic binding case except that processes must also be mutually non-interfering with respect to replacable objects.

PROGRAM CERTIFICATION UNDER STATIC BINDING

Chapter 3

In Section 2.4.4 we specified the security requirements of a multiclass system with static binding of security classes. We examined a run-time mechanism proposed by Fenton [Fe73,Fe74a] that guarantees the secure execution of a program in this environment, and we stated that security can also be guaranteed with a compile-time mechanism that certifies the secure execution of a program before it executes. The purpose of this chapter is to present such a mechanism.

Program certification mechanisms have several advantages over run-time verification mechanisms in protection systems. One important advantage is that the execution of a program is guaranteed to be secure <u>before</u> it executes; hence a program cannot leak information by purposely causing security violations (see Section 2.2). Another is that it does not impair the execution speed of a program, since all security checks are performed prior to program execution (although some run-time support may still be necessary or desirable for reliability reasons). Of course this would be of dubious value if the certification mechanism added substantially to the cost of compilation! The mechanism proposed here, however, does not and indeed adds suprisingly little overhead to the compilation process. A third advantage is that the certification process itself can be specified in terms of higher level language structures, rather than low level hardware instructions. In this form it is more understandable, and correctness is more easily established.

The principles of program certification were first formalized by Floyd [F167] and Naur [Na66] in an effort to develop a methodology for proving program correctness. The basic idea is that the programmer specifies "invariant assertions" about the state of the variables at various steps of the program. Program certification then is the process of proving that these assertions will hold for every possible execution of the program (i.e., for every set of input values). This can be done manually or (in principle at least) automatically by a theorem-proving program. Since it is a slow and tedious manual task, a considerable amount of research effort has been directed toward the construction of efficient automatic certification mechanisms. However, progress in this area has been slower than expected. A complete survey of techniques for proving program correctness has been given by Elspas et al. [EL72].

This chapter demonstrates the construction of an automatic compile-time mechanism that certifies the secure execution of a program under static binding of objects to security classes. In this case, only one form of assertion must be proved at each step in the program, namely, that the information α stored in a variable a satisfies the property $\underline{\alpha} \to \underline{a}$.

This is not the first time compiler certification mechanisms to verify security properties have been proposed. Conway et al. [CM72] investigated compile-time procedures for controlling a program's data independent accesses to objects in a data base (data dependent accesses are controlled by run-time procedures). Moore [Mo74] investigated a compile-time mechanism that constructs an "information flow graph" showing the flow of information between variables of a program and the conditions under which such flow occurs. Our results complement Moore's. In our case, the flow relation on the security classes is given, and the objective is to verify that the program satisfies these flow requirements. In his case, the objective is to construct a representation of possible flows among the variables of the program. Other related work in the general area of protection and programming languages can be found in [GS73,MC74,Mr73].

The approach we take is to show how to incorporate a certification mechanism into the analysis phase of a syntax-driven compiler. We do this by examining various syntactic constructs typical of high level languages (most of our constructs are taken from PASCAL [Wr71], Algol-60 [Na63], and PL/I [IB71]). For each syntactic type, we specify one or more actions to be performed (along with other semantic actions such as type checking and code generation) when a string of that type is recognized; these actions are referred to as the <u>certification semantics</u> of the language.

To control the complexity of the presentation, the mechanism will be introduced stepwise in six stages:

1) Expressions, assignment, and I/O of elementary data structures,
2) Conditional statements,
3) Gotos,
4) Interrupts,
5) Complex data structures, and
6) Procedure calls.

The objective of stage (1) is to certify all explicit flows among simple data objects. Implicit flows, such as those arising from conditional statements and gotos, are deferred to the later stages (see Section 2.4.4 for a discussion of implicit v. explicit flow). At stage (1) it is also necessary to assume that programs are uninterruptible — i.e., their execution cannot be interrupted for exceptional conditions such as end-file or overflow. The reason is that the occurrence of an interrupt may cause the subsequent execution of statements to be conditioned on the data that caused the interrupt and thereby result in an implicit flow.

At stage (2) we consider the certification of implicit flows that arise during the execution of three common conditional control statements: if-then-else, while-do, and repeat-until. As will be seen, the technique used actually applies to any conditional statement, such as the PASCAL case statement [Wr71] or the Algol for statement [Na63]. At this stage, we are therefore able to certify simple "structured programs".

At stage (3) we consider the rather unattractive complications begotten by the allowance of a goto statement. Since the goto statement permits unrestricted transfer of control, locating all statements conditioned on a particular expression becomes considerably more difficult. Indeed, it is necessary to perform a complete control flow analysis of a program before its security can be certified. Since the use of unrestricted gotos is of questionable value anyway, we do not consider all of the intricacies of therequisite control flow analysis. The purpose of this analysis is to illustrate the general principle involved in certifying general control flow structures. The fact of its excessive complexity further justifies the practicality of the simpler, structured approach taken at stage (2).

At stage (4) we suggest two viable approaches for guaranteeing security in an environment in which interrupts can occur. The first approach — program structuring — requires a program to conform to a rather restricted structure: it must assign values to security classes in a monotonically increasing order (i.e., it must first compute all its lowest security results, then its next lowest, and so on). The second approach — interrupt inhibition — permits a program to define its own "interrupt handlers" with an on-statement and inhibits all interrupts for which no such action is defined. This latter approach allows greater flexibility than the former, though it does require modest run-time support.

At stage (5) we consider three complex data structures: arrays, records, and lists. Whereas the elements of a record can belong to different security classes, the elements of arrays and lists must belong to the same class, since in the latter cases there is no way of determining at compile-time which element is being referenced.

At stage (6) we consider procedure calls and observe that it is not generally possible for the compiler to complete the certification of a program which calls external procedures. However, the compiler can generate information which when subsequently used by a linker (i.e., a linkage editor or linking loader), will verify that the collection of subroutines bound together is secure. In the special case where a call is made to an external procedure that derives all of its results from the input parameters and data in the lowest class L, the compiler can complete the verification of the procedure call.

3.1 Expressions, Assignment, and I/O of Elementary Data Structures

Consider a language which allows only the elementary data types known as "constants" – i.e., integers, reals, characters, and Booleans – and which allows only the elementary data objects known as "scalar variables" and "files" of the elementary data types. All constants are assumed to belong to the lowest security class L, and all variables and files are assumed to be assigned to a security class by an explicit declaration, such as

a: integer of class A.

(Observe that we are in effect treating security classes as data types.) Arithmetic and Boolean expressions are formed from these data structures in the usual way.

We shall employ the notation $\underline{\langle exp\rangle}$ to denote the security class of an expression; for example, the security class of the expression "a * b + c" is $\underline{\langle exp\rangle} = \underline{a} \oplus \underline{b} \oplus \underline{c}$. In general, we denote by $\underline{\langle x\rangle}$ the security class associated with a string of syntactic type <x>.

There are only three statement types to consider at this point: assignment, input, and output. An assignment statement has the structure

<var> := <exp> .

Since this statement represents a flow from <exp> to <var>, security requires that

$\underline{\langle exp\rangle} \rightarrow \underline{\langle var\rangle}$.

For example, the statement "c := a + b" has the structure

<exp> = "a + b" and <var> = "c", with corresponding security requirement $\underline{a} \oplus \underline{b} \rightarrow \underline{c}$. An input statement has the structure

input <inlist> from <file>

where <inlist> is a list of input variables $\text{<var>}_1, \ldots, \text{<var>}_n$. Since this statement represents a flow from <file> to <var>_i $(1 \leq i \leq n)$, security requires that

$\underline{\text{<file>}} \rightarrow \underline{\text{<var>}}_i \quad (1 \leq i \leq n)$.

Letting $\underline{\text{<inlist>}} = \underline{\text{<var>}}_1 \otimes \ldots \otimes \underline{\text{<var>}}_n$, (by Proposition 2.3-5) this requirement is equivalent to

$\underline{\text{<file>}} \rightarrow \underline{\text{<inlist>}}$.

For example, the statement "input a, b from c" has the structure <file> = "c" and <inlist> = "a, b", with corresponding security requirement $\underline{c} \rightarrow \underline{a} \otimes \underline{b}$. An output statement has the structure

output <outlist> to <file>

where <outlist> is a list of expressions $\text{<exp>}_1, \ldots, \text{<exp>}_n$. Since this statement represents a flow from <exp>_i to <file> $(1 \leq i \leq n)$, security requires that

$\underline{\text{<exp>}}_i \rightarrow \underline{\text{<file>}} \quad (1 \leq i \leq n)$.

Letting $\underline{\text{<outlist>}} = \underline{\text{<exp>}}_1 \oplus \ldots \oplus \underline{\text{<exp>}}_n$, (by Proposition 2.3-2) this becomes

$\underline{\text{<outlist>}} \rightarrow \underline{\text{<file>}}$.

For example, the statement "output a, b*c to d" has the structure <outlist> = "a, b*c" and <file> = "d", with corresponding security requirement $\underline{a} \oplus \underline{b} \oplus \underline{c} \rightarrow \underline{d}$.

Syntax and certification semantics for the above statement types is summarized in Figure 3.1-1. If a rule contains two occurrences of a

```
Syntax                                              Certification Semantics
<addop> ::= + | - | v
<mulop> ::= * | / | ^
<relop> ::= = | ≠ | < | ≤ | > | ≥

<factor> ::= <var>                                  <factor> := <var>
<factor> ::= <const>                                <factor> := L
<factor> ::= ( <exp> )                              <factor> := <exp>
<factor> ::= ¬ <factor>'                            <factor> := <factor>'

<term> ::= <factor>                                 <term> := <factor>
<term> ::= <term>' <mulop> <factor>                 <term> := <term>' ⊕ <factor>

<aexp> ::= <term> | <addop> <term>                  <aexp> := <term>
<aexp> ::= <aexp>' <addop> <term>                   <aexp> := <aexp>' ⊕ <term>

<exp> ::= <aexp>                                    <exp> := <aexp>
<exp> ::= <aexp> <relop> <aexp>'                    <exp> := <aexp> ⊕ <aexp>'

<inlist> ::= <var>                                  <inlist> := <var>
<inlist> ::= <inlist>' , <var>                      <inlist> := <inlist>' ⊗ <var>

<outlist> ::= <exp>                                 <outlist> := <exp>
<outlist> ::= <outlist>' , <exp>                    <outlist> := <outlist>' ⊕ <exp>

<assign stmt> ::= <var> := <exp>                    if <exp> ≰ <var> then SECURITY ERROR

<input stmt> ::= input <inlist> from <file>         if <file> ≰ <inlist> then SECURITY ERROR

<output stmt> ::= output <outlist> to <file>        if <outlist> ≰ <file> then SECURITY ERROR
```

Figure 3.1-1. Syntax and Certification Semantics for Assignment and I/O.

syntactic type <x>, we denote the second occurrence by <x>' so that the semantics are not ambiguous. The interpretation of the certification semantics for a rule

<x> ::= ω

where ω is a string of terminals and non-terminals is straightforward. When a string of form ω is recognized – i.e., a "syntax tree" with root node <x> is constructed, the action specified by the semantics is performed. If this action includes an assignment of the form <$\underline{x}$> := c, where c is a security class, then the class c (derived from nodes of subtrees of <x>) is associated with the root node <x>. If the action includes a security test and that test fails, the security error is reported and the program is not certified. The reader unfamiliar with such specifications may wish to consult Gries [Gi71: Section 12.2] for similar semantic routines that transform infix expressions to quadruples. The implementation of the parser and certification routines is not important (though clearly, some parsing techniques are more suitable than others).

To illustrate the certification process, we consider a bottom-up parse of the assignment statement "d := a * b + c". As the expression on the right side is recognized, the classes of the operands a, b, and c are combined using ⊕. This proceeds as follows: when "a * b" is reduced to a <term>, $\underline{a} \oplus \underline{b}$ is assigned to <$\underline{\text{term}}$>; then when "<term> + c" is reduced to an <exp>, <$\underline{\text{term}}$> ⊕ $\underline{c}$ is assigned to <$\underline{\text{exp}}$>. The security of the assignment statement is easily verified when "<var> := <exp>" is reduced to an <assign stmt> by testing <$\underline{\text{exp}}$> → <$\underline{\text{var}}$>. The details of the process are illustrated by Figure 3.1-2 which shows the syntax tree for

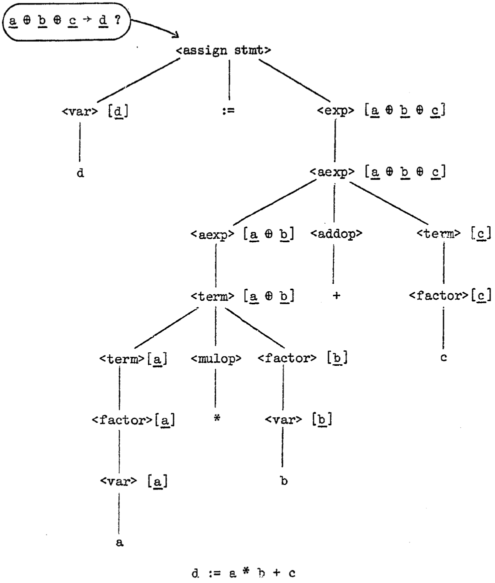


Figure 3.1-2. Certification Tree of an Assignment Statement.

the expression; the security classes (in brackets [ ]) have been associated with each node of the tree, and the nodes at which flows must be verified are indicated. Such a tree is defined to be a *certification tree*.

The certification of *input* and *output* statements is similar. In the case of an *input* statement, the variables constituting the input list are combined using $\otimes$ as the reduction to <inlist> is performed. Figure 3.1-3 displays the certification tree for the statement "*input* a, b *from* c". In the case of an *output* statement, the expressions constituting the output list are combined using $\oplus$ as the reduction to <outlist> is performed.

3.2 Conditional Statements

We now extend the syntax and the corresponding certification semantics described in the previous section to include three conditional statements:

if <exp> *then* <stmt>$_1$ [*else* <stmt>$_2$]

while <exp> *do* <stmt>$_1$

repeat <stmt>$_1$ *until* <exp>

where <stmt> is an elementary statement (assignment or I/O), conditional statement, or compound statement (*begin*-*end* statement). These statements all have the underlying abstract structure (see Section 2.4.4):

<exp>: <stmt>$_1$ [,<stmt>$_2$].

Since this structure specifies an implicit flow from <exp> to all variables and files assigned values in <stmt>$_1$ [and <stmt>$_2$], security requires (in addition to the independent security of the substatements) that all implicit flows from <exp> be secure. Letting V(<stmt>) denote

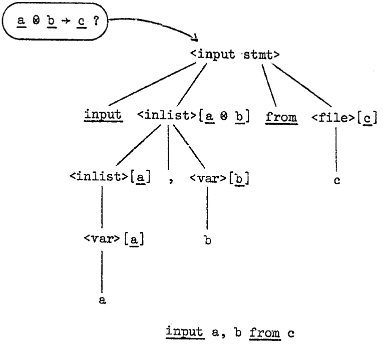


Figure 3.1-3. Certification Tree of an Input Statement.

the set of variables and files assigned values in a string of syntactic type <stmt> (see Section 2.4.4), this means that for a conditional statement <stmt> with structure

$$\langle\text{stmt}\rangle = \langle\text{exp}\rangle\text{: } \langle\text{stmt}\rangle_1 \ [, \ \langle\text{stmt}\rangle_2]$$

the following property must hold:

$$\underline{\langle\text{exp}\rangle} \rightarrow \underline{b} \quad \forall\, b \in V(\langle\text{stmt}\rangle_1)\ [\cup V(\langle\text{stmt}\rangle_2)]$$

or equivalently:

$$\underline{\langle\text{exp}\rangle} \rightarrow \underline{b} \quad \forall\, b \in V(\langle\text{stmt}\rangle).$$

For if-then-else and while-do statements, this property could be verified during the analysis of $\langle\text{stmt}\rangle_1$ [and $\langle\text{stmt}\rangle_2$] since <exp> is recognized before these substatements are parsed. However, this technique is not suitable for the repeat-until statement since <exp> is recognized after $\langle\text{stmt}\rangle_1$ is analyzed. Therefore, it is necessary to verify the above property after the complete conditional statement is recognized.

To see how this can be done, observe that (by Proposition 2.3-5) the above property is equivalent to

$$\underline{\langle\text{exp}\rangle} \rightarrow \otimes\{\underline{b} \mid b \in V(\langle\text{stmt}\rangle)\}.$$

This suggests a set of actions that forms $\otimes\{\underline{b} \mid b \in V(\langle\text{stmt}\rangle)\}$ for each statement (non-conditional as well as conditional) as it is recognized. These actions, shown in Figure 3.2-1, supplement the actions specified in Figure 3.1-1. In Figure 3.2-1 we have used $\underline{\langle\text{stmt}\rangle}$ to denote the class $\otimes\{\underline{b} \mid b \in V(\langle\text{stmt}\rangle)\}$ since no other class was previously associated with <stmt>.

The alert reader may recognize a syntactic ambiguity in the grammar for an <if stmt>; this does not affect our analysis and can be dealt with by the usual techniques of resolving ambiguities in programming language syntax [Gi71].

Syntax	Certification Semantics
<assign stmt> ::= <var> := <exp>	<assign stmt> := <var>
<input stmt> ::= <input <inlist> from <file>	<input stmt> := <inlist>
<output stmt> ::= output <outlist> to <file>	<output stmt> := <file>
<if stmt> ::= if <exp> then <stmt>	<if stmt> := <stmt> if <exp> ↛ <if stmt> then SECURITY ERROR
<if stmt> ::= if <exp> then <stmt> else <stmt>'	<if stmt> := <stmt> ⊗ <stmt>' if <exp> ↛ <if stmt> then SECURITY ERROR
<while stmt> ::= while <exp> do <stmt>	<while stmt> := <stmt> if <exp> ↛ <while stmt> then SECURITY ERROR
<repeat stmt> ::= repeat <stmt> until <exp>	<repeat stmt> := <stmt> if <exp> ↛ <repeat stmt> then SECURITY ERROR
<stmt list> ::= <stmt>	<stmt list> := <stmt>
<stmt list> ::= <stmt list>' ; <stmt>	<stmt list> := <stmt list>' ⊗ <stmt>
<compound stmt> ::= begin <stmt list> end	<compound stmt> := <stmt list>
<stmt> ::= <assign stmt>	<stmt> := <assign stmt>
<stmt> ::= <input stmt>	<stmt> := <input stmt>
<stmt> ::= <output stmt>	<stmt> := <output stmt>
<stmt> ::= <if stmt>	<stmt> := <if stmt>
<stmt> ::= <while stmt>	<stmt> := <while stmt>
<stmt> ::= <repeat stmt>	<stmt> := <repeat stmt>
<stmt> ::= <compound stmt>	<stmt> := <compound stmt>

Figure 3.2-1. Syntax and Certification Semantics for Conditional Statements.

The certification process is illustrated in Figure 3.2-2 for the statements

```
a)   while a > 0 do
        begin
           b := b + 1;
           output b to c;
           a := a - 1
        end

b)   if a = 0
        then while b < c do b := b + 1
        else d := 1.
```

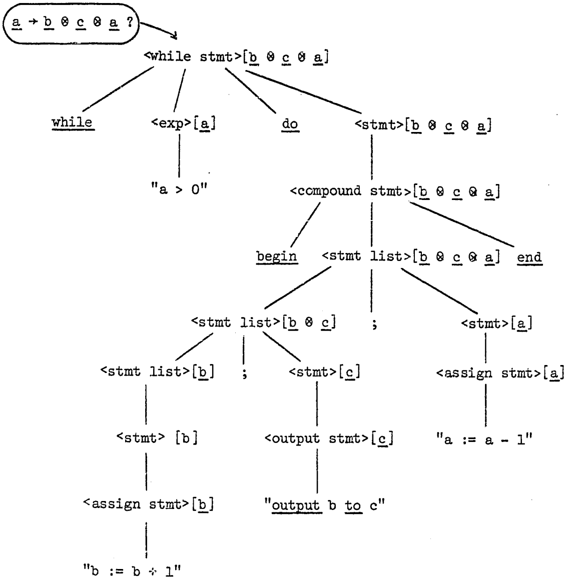
It should be clear that this certification process is equally applicable to all statements with an underlying abstract structure

<exp>: <stmt>$_1$,...,<stmt>$_n$ $(n \geq 1)$.

This includes case statements, for statements, etc. (see also Section 2.4.4).

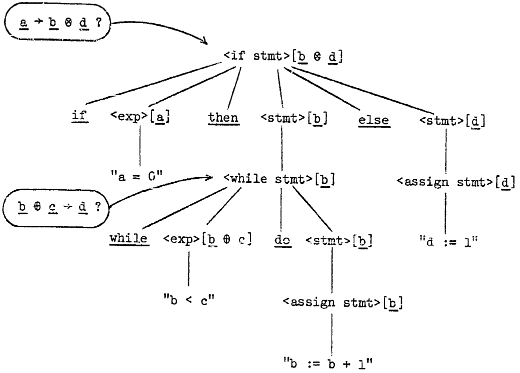
3.3 Gotos

If the language is extended to include a goto statement, the task of certifying implicit flows becomes considerably more complex. This is because goto statements give rise to arbitrary control flow structures that are not isolated by syntactic elements, making it more difficult to determine the set of statements immediately conditioned on an expression. To illustrate, consider the program in Figure 3.3-1. Because of the extensive use of gotos, it is practically impossible to tell which statements are conditioned on the test "b = 0" at the statement labelled "4". In general, with gotos, the set of statements conditioned on an expression may be scattered throughout an entire program. (It is interesting to note that these considerations forced Fenton to introduce a "call" instruction into his Minsky machine, so that the



a) while a > 0 do begin b := b + 1; output b to c; a := a - 1 end

Figure 3.2-2. Certification Trees of Conditional Statements.



b) if a = 0 then while b < c do b := b + 1 else d := 1

Figure 3.2-2, cont.

Program	Partitioning into Basic Blocks

1: input a, b from f1;
c := 0;
if a = 0 then goto 4;
	Φ_1; ξ_1 = "a = 0"

2: a := a + 1;	Φ_2; ξ_2 = null
3: output a to f2; if a < b then goto 2 else goto 6;	Φ_3; ξ_3 = "a < b"
4: if b = 0 then goto 3;	Φ_4; ξ_4 = "b = 0"

5: b := a;
c := 1;
	Φ_5; ξ_5 = null

6: output a, b to f2	Φ_6; ξ_6 = null

Figure 3.3-1. Example of a Program with Gotos.

scope of a condition could be limited — see Section 2.4.4) For this reason, it is not possible to certify implicit flows until the entire program is analyzed and a control flow analysis performed. Only then can the underlying abstract conditional structures

$\xi: \Phi_1,\ldots,\Phi_m$

be determined, and implicit flows from the expression ξ to the objects (i.e., variables and files) assigned values in the program substructures $\Phi_1,\ldots,\Phi_m$ be verified.

We now outline a general procedure for performing this control flow analysis and verifying the security of all implicit flows. This analysis is performed _not_ because we believe the mechanism should support _gotos_, but _rather_ to illustrate the general characteristics of all conditional structures and to justify the approach taken in the preceding section.

The first step is to partition the program into a set of substructures $\Phi_1,\ldots,\Phi_n$ each of which is a basic block. A _basic block_ is a sequence of statements with exactly one entry point (the first statement in the sequence) and exactly one exit point (the last statement in the sequence). It is important to note that each basic block Φ_i contains at most one conditional statement (e.g., _if-then-else_ statement), which must be the last statement of Φ_i. The conditional expression in this statement (if it exists) will be denoted ξ_i. Figure 3.3-1 shows the partitioning for the sample program. A thorough treatment of basic blocks and the control flow analysis to be described can be found in [Al70,LM69,Ta74].

Let $V(\Phi)$ denote the set of variables assigned values in basic block Φ. Associate with each block Φ a class $\underline{\Phi}$,

$$\underline{\Phi} = \begin{cases} \otimes\{\underline{b} \mid b \in V(\Phi)\} & \text{if } V(\Phi) \neq \emptyset \\ H & \text{otherwise} \end{cases}$$

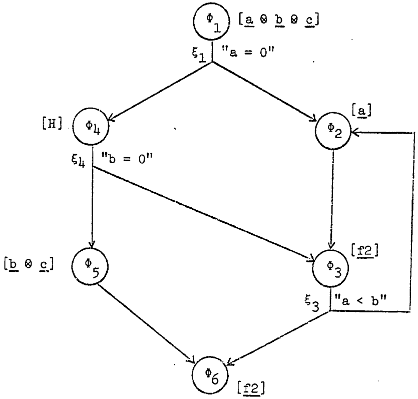
where H is the highest security class and therefore an identity element over $\otimes$. Note that the process of partitioning a program into basic blocks and of associating classes with the basic blocks can be performed during the analysis phase of compilation, using techniques similar to those described in the preceding section.

The next step is to construct a control flow graph of the program; that is, a directed graph whose nodes are the basic blocks and whose edges represent the control flow paths. There is an edge from Φ_j to Φ_k if control can flow directly from Φ_j to Φ_k. Figure 3.3-2 shows the control flow graph of the sample program of Figure 3.3-1. Here we have labelled each node Φ with its class $\underline{\Phi}$ (in brackets [ ]). For those nodes with more than one outgoing edge, we have also shown the conditional expression that determines which edge will be selected.

The final step is to use the control flow graph to determine the set of basic blocks (nodes) conditioned on each expression ξ_i. Let Ψ_i denote the set of basic blocks conditioned on ξ_i. All implicit flow from ξ_i is then certified for security by verifying that

$$\underline{\xi_i} \rightarrow \otimes\{\underline{\Phi} \mid \Phi \in \Psi_i\}.$$

For basic block Φ_i associated with expression ξ_i, the immediate forward dominator IFD(Φ_i), is the closest block to Φ_i among the set of all blocks lying on every path from Φ_i to the exit block of the graph.



$\mathrm{IFD}(\Phi_1) = \Phi_6$ $\Psi_1 = \{\Phi_2, \Phi_3, \Phi_4, \Phi_5\}$

$\mathrm{IFD}(\Phi_3) = \Phi_6$ $\Psi_3 = \{\Phi_2, \Phi_3\}$

$\mathrm{IFD}(\Phi_4) = \Phi_6$ $\Psi_4 = \{\Phi_2, \Phi_3, \Phi_5\}$

Figure 3.3-2. Control Flow Graph of Program of Figure 3.3-1.

It follows that

$$\Psi_i = \{\Phi \mid \Phi \text{ lies on a path from } \Phi_i \text{ to } \mathrm{IFD}(\Phi_i)\}.$$

This follows from the observation that all control flow paths originating with Φ_i join at IFD(Φ_i); hence, the only blocks directly conditioned on ξ_i are those that lie on some path from Φ_i to IFD(Φ_i). (See Figure 3.3-2.) Of course, a block Φ executed after IFD(Φ_i) can be indirectly conditioned on ξ_i if an assignment is made to some variable b in $\Phi' \varepsilon \Psi_i$, and Φ is directly conditioned on the value of b. Because the flow relation is transitive, we do not have to worry about such indirect conditioning: the flows from ξ_i to b and from b to all variables assigned values in Φ will be verified.

It is now easy to justify the mechanism of the preceding section for certifying the conditional statements

if <exp> then <stmt>$_1$ [else <stmt>$_2$],

while <exp> do <stmt>$_1$, and

repeat <stmt>$_1$ until <exp>

since in all cases, the immediate forward dominator (of $\Phi_{\text{<exp>}}$) is precisely the next statement, and therefore $\Psi_{\text{<exp>}}$ is precisely the conditional statement itself. Hence, it is possible to determine the underlying abstract conditional structures of these statements without performing a global flow analysis.

3.4 Interrupts

Additional mechanisms are needed to guarantee security in the presence of interrupts. This is because an interrupt may cause the execution of subsequent statements to be conditioned on the data that caused the interrupt, giving rise dynamically to implicit flows that go undetected by our present mechanism. Consider, for example, the execution of the following program in an environment where an overflow interrupt can result in program termination:

```
sum := 0;
i := 0;
repeat
   begin
      sum := sum + x;
      i := i + 1;
      output i to b
   end
until false
```

Assume $\underline{i} = \underline{b} = L$ (the lowest security class) and $\underline{x} = \underline{sum} = H$ (the highest security class). Our present mechanism would certify this program, since the iteration appears to be conditioned on the Boolean constant false which is associated with L, and $L \rightarrow \underline{i} \otimes \underline{sum} \otimes \underline{b}$. Should the variable sum overflow, however, the iteration would be terminated by the overflow interrupt, whereupon the iteration will have been conditioned on sum and an invalid flow from sum to b will have occurred (since $\underline{sum} = H \not\rightarrow \underline{b} = L$). Indeed, in this case file b contains information about x: the value of x can be approximated by computing MAXVALUE/LASTb, where MAXVALUE is a constant representing the largest value a variable could assume before overflowing, and LASTb is the last value of i written into file b. However, this violates security since $\underline{x} \not\rightarrow \underline{b}$. Our present certification mechanism would have detected this had the programmer made explicit his intentions:

```
repeat
   begin
      sum := sum + x;
      i := i + 1;
      output i to b
   end
until sum overflows
```

There appear to be four general approaches to solving this problem. The first two are too drastic to be viable; the third reduces the problem to the single-class case of Section 2.4.1; and the fourth, which inhibits all interrupts except those for which the programmer has specified explicit actions, is the most attractive in the context of the mechanism under consideration in this chapter. Each will be considered below.

The first possible solution is a run-time mechanism that receives control when a program interrupts. It would abort the program, erase its entire memory (including all files it had written), and "sound an alarm". This solution is unacceptable for at least two reasons: first, there is no good way of determining at run-time whether or not an interrupt caused a security violation; and second, a program should be able to process, without "sounding an alarm", any interrupt that does not cause a violation.

The second possible solution is based on the principle of restructuring a program so that the certification mechanism can verify the correctness of interrupts. The idea is to have the compiler insert conditional expressions, corresponding to conditions that might cause interrupts, at the appropriate points in the program. These conditions are then used to verify the security of all interrupts. For example, the preceding program could be modified to:

```
sum := 0;
i := 0;
repeat
   begin
      if sum + x ≤ MAXVALUE then
         begin
            sum := sum + x;
            if i + 1 ≤ MAXVALUE then
               begin
                  i := i + 1;
                  output i to b
               end
            else halt
         end
      else halt
   end
until false
```

The compiler would now detect the security violation in this program since the dependency of i and b on x is explicit. This solution is unattractive because it seriously complicates the compilation process, possibly to the point of rendering the entire certification mechanism too cumbersome to be useful.

The third possible solution is based on the principle of imposing so many restrictions on program structure that interrupts cannot cause the problem discussed above. One method of doing this is to require that the program be specified as a collection of modules, in one-to-one correspondence with the security classes. If Φ_A is the program module corresponding to class A, the following constraints must be observed:

a) Φ_A may assign values only to objects in class A;

b) If x is an input parameter to Φ_A, then $\underline{x} \rightarrow A$;

c) If y is an output parameter from Φ_A, then $A \rightarrow \underline{y}$;

d) If Φ_A transfers control to Φ_B, then $A \rightarrow B$.

Under these restrictions, an interrupt occurring in Φ_A can be conditioned only on those objects z for which $\underline{z} \rightarrow A$, so that the problem is eliminated. It should be noted that these restrictions make the problem isomorphic to the single-class case considered in Section 2.4.1, where files are used to transmit parameters among processes. In this case there is no need for compiler certification of the modules, so we shall not consider it further.

The fourth solution is based on the simple observation that no security violation can occur in an environment that inhibits interrupts, since the only way a statement can in this case be conditioned on the state of a variable is by explicitly testing its value. However, a completely interrupt-free environment is impractical, as a program should be allowed to process certain interrupts such as end-file or overflow, should it desire to do so. We therefore propose to 1) extend the language to include a construct for specifying interrupt conditions and procedures for handling them ("interrupt handlers"), 2) modify the certification process to certify programs in the extended language, and 3) include a run-time mechanism that invokes (as a procedure) the interrupt handler of a specified interrupt but inhibits an unspecified interrupt.

The language construct we add is similar to the PL/I ON statement [IB71], and has the following structure:

on <condition> <ident> do <stmt>,

where <condition> names an interrupt condition (e.g., overflow, zero-divide, or end-file), <ident> is the variable or file name to which it applies, and <stmt> is the statement to be called when the interrupt occurs (after execution of the statement, control returns to the place where the interrupt occurred). Examples of on statements are:

```
on endfile b do moredata := false

on overflow x do begin x := MAXVALUE; flag := true end
```

The scope of an on statement is not important; i.e., it could be a single statement, a block, or the entire program.

It should be clear from our preceding analysis of conditional structures (Sections 3.2 and 3.3) that the on statement is just another example of a conditional statement; hence the security requirements are simply

```
<var> → <stmt>,
```

which leads to the certification semantics

```
<on stmt> := <stmt>
 if <var> ↛ <on stmt> then SECURITY ERROR.
```

To illustrate the mechanism, consider again the program that divulged the value of variable x by causing the variable sum to overflow. In the present context, failure to specify an on statement for the condition "sum overflows" will not enable this interrupt, whereupon the program will go on executing forever (or until the program exhausts its ration of processor time). In this case, the last value of i written into file b contains no information whatsoever about x. On the other hand, if the overflow condition is specified explicitly:

```
on overflow sum do flag := false;
flag := true;
sum := 0;
i := 0;
repeat
   begin
      sum := sum + x;
      i := i + 1;
      output i to b
   end
until ¬ flag
```

the security violation will be detected when the compiler verifies the conditions $\underline{\text{sum}} \rightarrow \underline{\text{flag}}$ and $\underline{\text{flag}} \rightarrow \underline{\text{sum}} \otimes \underline{\text{i}} \otimes \underline{\text{b}}$.

This last solution is the most attractive, since it permits a program to process selected interrupts without requiring substantial overhead or imposing undue restrictions on program structure.

3.5 Complex Data Structures

3.5.1 Arrays

Let a be an array having n dimensions (subscripts). The elements of a must all belong to the same security class, since there is no way of determining at compile-time which element (or even which row or column) is referenced by an expression like $a[i_1,\ldots,i_n]$. Therefore, an array is bound to a single class.

The certification semantics for array references, shown in Figure 3.5-1, are based on the principle that an array reference

<array> [<sublist>]

is secure only if

$\underline{\text{<sublist>}} \rightarrow \underline{\text{<array>}}$,

where

$\underline{\text{<sublist>}} = \underline{\text{<exp>}}_1 \oplus \ldots \oplus \underline{\text{<exp>}}_n$

and $\text{<exp>}_1,\ldots,\text{<exp>}_n$ are the subscripts constituting <sublist>.

The semantics for the rule "<array ref> ::= <array> [<sublist>]" include the generation of code to check that <sublist> specifies subscripts within the range of <array>. Without this check, it would be possible for a statement like "a[i] := b" to cause an invalid from variable b to a variable c if i is out of range (i.e., a[i] references

Syntax	Certification Semantics
<sublist> ::= <exp>	<sublist> := <exp>
<sublist> ::= <sublist>' , <exp>	<sublist> := <sublist>' ⊕ <exp>
<array ref> ::= <array> [<sublist>]	if <sublist> ↛ <array> then SECURITY ERROR <array ref> := <array> generate code to check range of <sublist>
<var> ::= <simple var>	<var> := <simple var>
<var> ::= <array ref>	<var> := <array ref>

Figure 3.5-1. Syntax and Certification Semantics for Array References.

variable c rather than an element of a). In case the programmer has specified actions for handling invalid subscripts, the methods of Section 3.4 can be used to verify the security of the corresponding interrupt. In case the programmer has not specified such actions, the compiler can generate code that causes the invalid reference to refer to the first element of the array, for example, so that no invalid flow can occur.

Note that we have also defined (Figure 3.5-1) the syntactic type <var> to be a simple variable or array reference now. Whereas <array ref> is determined when the structure is recognized, <simple var> is obtained directly from the symbol table.

It should be noted that, in principle, there is a slight difference between array references appearing in an expression and those appearing on the left side of an assignment statement. In the former case, it is necessary only that

<array ref> := <array> ⊕ <sublist>

but not that

<sublist> → <array>

(which is a stronger requirement). We did not consider this distinction important enough to present the more complicated semantics required to identify whether the array reference is the left part of an assignment or not. However, the semantics of Figure 3.5-1 can be extended to this case if need be.

3.5.2 Records

Define a record r to be a structure consisting of n elements or fields, $x_1,\ldots,x_n$, the element x_i being identified by the compound name

$r.x_i$. Unlike an array, the elements of a record can belong to different security classes, since they are always referenced by unique names. In fact, record elements can be treated exactly as other variables of syntactic type <simple var>. However, doing so requires some care in the specification of the certification semantics for records (see Figure 3.5-2).

The statement "<u>input</u> r <u>from</u> a" where a is a file evidently requires $\underline{a} \rightarrow \underline{r.x_i}$ for $1 \leq i \leq n$, or equivalently that

$$\underline{a} \rightarrow \underline{r.x_1} \otimes \ldots \otimes \underline{r.x_n} \equiv \bigotimes \underline{r}.$$

Similarly, the statement "<u>output</u> r <u>to</u> b" requires that

$$\bigoplus \underline{r} \equiv \underline{r.x_1} \oplus \ldots \oplus \underline{r.x_n} \rightarrow \underline{b}.$$

An assignment "r := s", which is valid syntactically only if r and s have identical structure, is secure only if

$$\underline{s.x_i} \rightarrow \underline{r.x_i} \quad (1 \leq i \leq n).$$

Note that the apparently simpler check $\bigoplus \underline{s} \rightarrow \bigotimes \underline{r}$ does not work in this case since, for example, it will reject assignments "r := s" in which $\underline{r.x_i} = \underline{s.x_i}$ for all i but $\underline{s.x_i} \neq \underline{s.x_j}$ for some $i \neq j$.

3.5.3 List Structures

Define a list structure ℓ to be a set of one or more records of identical structure. A record in ℓ is referenced by a pointer variable p, associated with ℓ in the declaration. We denote by $\ell \uparrow p$ the record of ℓ pointed to by p, and by $\ell.x\uparrow p$ the element x in the record of ℓ pointed to by p.

The security requirements are a combination of those for arrays and records. As for arrays, a pointer p can select an arbitrary member from the aggregate; therefore, it is necessary that $\underline{\ell.x\uparrow p}$ be a single class,

Syntax	Certification Semantics
<rec input stmt> ::= input <record> from <file>	<record input stmt> := $\otimes$<record> if <file> $\not\rightarrow$ <rec input stmt> then SECURITY ERROR
<rec output stmt> ::= output <record> to <file>	<rec output stmt> := <file> if $\oplus$<record> $\not\rightarrow$ <rec output stmt> then SECURITY ERROR
<rec assign stmt> ::= <record> := <record>'	if <record> and <record>' have corresponding elements $x_1,\ldots,x_n$, then { if $\exists$i (1≤i≤n): <record>'.x_i $\not\rightarrow$ <record>.x_i then SECURITY ERROR <rec assign stmt> := $\otimes$<record> else TYPE ERROR

Figure 3.5-2. Syntax and Certification Semantics for Records.

irrespective of the value of p, for any given field x. Also, the security of the reference $\ell.x\uparrow p$ requires that $\underline{p} \rightarrow \underline{\ell.x}$, and the security of $\ell\uparrow p$ requires $\underline{p} \rightarrow \bigotimes \underline{\ell}$. Since $\underline{p} \rightarrow \bigotimes \underline{\ell}$ implies $\underline{p} \rightarrow \underline{\ell.x}$ for each field x, verifying that $\underline{p} \rightarrow \bigotimes \underline{\ell}$ is sufficient for all references to elements of list ℓ. In addition to these security requirements for record references, the security requirements for flows to and from the records of ℓ (see Figure 3.5-2) apply here also. Because of these similarities we have not shown the certification semantics for list structures in a separate figure. As with arrays, it is necessary to generate code that checks whether the pointer p always designates an element of the list ℓ, after each assignment to p.

3.6 Procedure Calls

Let f be a procedure with formal input parameters $x_1,\ldots,x_m$ and formal output parameters $y_1,\ldots,y_n$. Consider an elementary statement of the form

<u>call</u> $f(a_1,\ldots,a_m;\ b_1,\ldots,b_n)$,

where $a_1,\ldots,a_m$ are the actual input parameters of f, and $b_1,\ldots,b_n$ are the actual output parameters of f. It is easy to see that the security requirements of this statement are simply:

a) f must be secure,

b) $\underline{a}_i \rightarrow \underline{x}_i \quad (1 \leq i \leq m)$, and

c) $\underline{y}_j \rightarrow \underline{b}_j \quad (1 \leq j \leq n)$.

As usual, if the <u>call</u> statement is conditioned on an expression ξ, it is necessary to verify the implicit flow from ξ to the output parameters:

d) $\underline{\xi} \rightarrow \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$.

For the moment we assume f does not write into any objects that are global or have a "lifetime" exceeding the activation of f (e.g., Algol <u>own</u> variables [Na63]), so that there are no other implicit flows from an expression ξ as a result of the call to procedure f. Later we show how to remove this restriction. Now, condition (d) can be verified by the same technique used to verify any implicit flow, namely by computing $\underline{b}_1 \otimes \ldots \otimes \underline{b}_n$ as the output parameters are recognized and associating this with the representative statement type, <proc call stmt> (see Figure 3.6-1). However, conditions (a) - (c) cannot be verified until after procedure f is compiled and its security established. Hence, it is necessary to save the classes of the parameters (see Figure 3.6-1) and check these conditions at the time f is linked to the calling program. To handle the case where f is compiled independently, it is therefore necessary to extend the certification mechanism to include a mechanism in the linker for certifying explicit flows not "bound" at compile-time. This implies that the certification of a program is not really complete until all external procedure calls are verified by the linker.

There is a serious limitation to the above approach when the procedure $f(x_1,\ldots,x_m;y_1,\ldots,y_n)$ is supposed to handle arbitrary classes of information (as is typical of library routines). Then $x_1,\ldots,x_m$ must be assigned to the highest security class H so that $\underline{a} \rightarrow \underline{x}_i$ $(1 \leq i \leq m)$ is guaranteed to hold for all calls. But this implies that $y_1,\ldots,y_n$ must be assigned to the highest security class H, since their values are derived from $x_1,\ldots,x_m$. This in turn requires that the caller of f must also assign $b_1,\ldots,b_n$ to H (even if $a_1,\ldots,a_m$ are in

```
Syntax                                                Certification Semantics
<input parms> ::= <exp> | <input parms> , <exp>       save  exp  for later verification

<output parms> ::= <var>                              <output parms> := <var>
                                                      save <var> for later verification
<output parms> ::= <output parms>' , <var>            <output parms> := <output parms>' ⊗ <var>
                                                      save <var> for later verification

<proc call stmt> ::= <proc name> ( <input parms>
                     ; <output parms> )               <proc call stmt> := <output parms>
```

Figure 3.6-1. Syntax and Certification Semantics for Procedure Calls.

```
Syntax                                                Certification Semantics
<input parms> ::= <exp>                               <input parms> := <exp>
<input parms> ::= <input parms>' , <exp>              <input parms> := <input parms>' ⊕ <exp>

<output parms> ::= <var>                              <output parms> := <var>
<output parms> ::= <output parms>' , <var>            <output parms> := <output parms>' ⊗ <var>

<proc call stmt> ::= <proc name> ( <input parms>      if <input parms> ≯ <output parms> then SECURITY ERROR
                     ; <output parms> )               <proc call stmt> := <output parms>

<fun call> ::= <proc name> ( <input parms> )          <fun call> := <input parms>

<factor> ::= <fun call>                               <factor> := <fun call>
```

Figure 3.6-2. Syntax and Certification Semantics for Restricted Procedure Calls.

the lowest class L!) so that $\underline{y}_j \rightarrow \underline{b}_j$ $(1 \leq j \leq n)$ also holds. By now the limitation should be obvious — it is impossible to obtain low security results from unrestricted procedures designed to handle data in arbitrary security classes.

One solution to this problem defines different versions of the procedure f for different security classes; the appropriate version is then selected at the time f is linked to the calling program. This is analogous to the GENERIC procedure in PL/I [IB71] which operates on different data types (i.e., integer, real, etc.).

Another solution restricts the procedure f so that the output parameters are derived entirely form the input parameters and data belonging to the lowest security class L (e.g., constants). If f is known to satisfy this property, then security can be established completely at compile-time by verifying

$$\underline{a}_1 \oplus \ldots \oplus \underline{a}_m \rightarrow \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$$

(see Figure 3.6-2). An interesting special case of such procedures is the "function" type procedure (SQRT, SIN, LOG, etc.). Here a procedure f is called during expression evaluation (by a call $f(a_1,\ldots,a_m)$) and returns with a single result, derived entirely from the input parameters $a_1,\ldots,a_m$ and constants. Since there are no output parameters, there are no side effects and no implicit flows. Figure 3.6-2 also shows syntax and certification semantics for such function calls.

Consider now the case where the procedure f writes into objects $z_1,\ldots,z_k$ that are global or have a "lifetime" exceeding the activation of f. For example, f could be a "monitor" procedure for controlling access to a shared message buffer [Ho74]. To guarantee the secure execution of f, it is now necessary to also verify implicit flows to $z_1, \ldots, z_k$ that result when a call to f is conditioned on one or more expressions $\xi_1, \ldots, \xi_r$. This can be done at the time f is linked to the calling procedure as follows. Associate with the procedure f an additional formal parameter z, where $\underline{z} = \underline{z_1} \otimes \ldots \otimes \underline{z_k}$, and with the call to f an additional actual parameter ξ, where $\underline{\xi} = \underline{\xi_1} \oplus \ldots \oplus \underline{\xi_r}$. The linking mechanism then verifies that $\underline{\xi} \rightarrow \underline{z}$.

3.7 Applications

The ability to certify programs and procedures for security using the methods of this chapter resolves several open questions about information security. It implies that it is possible to prevent leakage on many of the so-called "covert channels". It implies that the selective confinement problem has a solution. It implies that it is possible to design processes that can process confidential files, but cannot output information giving correlations between these files. It implies that data base query programs having security clearances can be properly designed. What is most important, all of the above can be handled with a simple, automatic certification procedure. These points are developed in the following subsections.

3.7.1 Covert Channels

The mechanism we have proposed will prevent leakage on any "stored information channel". By this we mean that if a process q is not permitted access to information in some class A, then q will be unable to obtain information in A by examining the contents of some storage object. This implies that it is possible to prevent leakage on many "covert channels". For example, a process p would not be able to leak a confidential value x by opening x files $a_1,\dots,a_x$ under the observation of a process q not authorized to see x (i.e., $\underline{x} \not\rightarrow \underline{q}$). To see why this is so, let b_i denote the "state variable" associated with file a_i; that is, b_i indicates whether or not a_i has been opened for reading or writing, etc.. We assume that $\underline{a_i} = \underline{b_i}$. If the certification mechanism certifies p, it must be true that $\underline{x} \rightarrow \underline{b_i}$ $(1 \leq i \leq x)$, since the value of each b_i depends on the value of x. But then q will not be able to access these variables as a means for determining the value of x unless $\underline{b_i} \rightarrow \underline{q}$ and, therefore, $\underline{x} \rightarrow \underline{q}$.

Of course this does not prevent p from leaking the value of x to q if p can alter the performance of the system in a way that is observable to q but not to a "security monitor". This is still an open problem.

3.7.2 Confinement

Lampson defines the confinement problem to be that of constraining a "service process" so that it cannot leak confidential information about a "customer process" [La73]. He outlines a solution to the problem that constrains the service process from retaining any information, confidential or not, after it ceases to operate on behalf of the customer process. This is also true of the solutions proposed in [An74, Jo73]. For this reason, the confined service process is referred to as "memoryless".

A more flexible solution to the problem would be to permit a confined service process to save non-confidential information in global variables and files. For example, a confined income tax computing service could then save address and billing information on its use by customers, though not information on its customers' incomes. Schroeder has suggested that a solution to this problem may be possible with a mechanism for certifying the process to be confined [Sc72]. We now show how this can be done with the mechanism proposed in this chapter. (Fenton also shows how this problem can be solved with his run-time mechanism [Fe74a]).

Let q(XN, XC; YN, YC) be a service process, where XN and XC are sets of input parameters for receiving non-confidential and confidential data respectively and YN and YC are sets of output parameters for returning non-confidential and confidential results respectively (XN, XC, YN, YC may contain single data elements, complex data structures, or files). Also let the information class of non-confidential data be A and that of confidential data be B (i.e., $x \in XN \Rightarrow \underline{x} = A$ and $x \in XC \Rightarrow \underline{x} = B$), where $A \rightarrow B$ but not $B \rightarrow A$. Then q is guaranteed to not leak confidential information about a customer p if it is secure and does not write into any global variables or files in a class C such that $B \nrightarrow C$. (We will assume its local memory is cleared and deleted when it terminates.) The methods of this chapter show that a compiler can certify q not only for security but also for confinement. Note that q may invoke other procedures, as long as they are also certified for confinement, and that the linkage to such procedures can be guaranteed secure.

The mechanism is also applicable to a more general case in which there may be more than two class of confidential information and the problem is to constrain a service process from leaking data belonging to some of these classes. Here we let $q(X_1,\ldots,X_n; Y_1,\ldots,Y_n)$ be a service process, where $X_1,\ldots,X_n$ are sets of input parameters for receiving data in classes $A_1,\ldots,A_n$ respectively and $Y_1,\ldots,Y_n$ are sets of output parameters for returning results in classes $A_1,\ldots,A_n$ respectively. Suppose that q is not permitted to leak data belonging to some set Z of classes (Z does not necessarily have to be a subset of $\{A_1,\ldots,A_n\}$). Then q may be certified for confinement if it is established that q is secure and does not write into any global variables or files in a class B such that $\bigotimes Z \rightarrow B$.

A problem where this more general confinement mechanism is useful is that of constraining a service process from leaking any information that links information in two classes A_1 and A_2 together (i.e., information derived from data in both A_1 and A_2) while permitting it to retain information in classes A_1 and A_2 separately. This can be achieved by letting $Z = \{A_1 \oplus A_2\}$, so that the confined process cannot write into any class B such that $A_1 \oplus A_2 \rightarrow B$. For example, A_1 may contain a list of names and A_2 a list of salaries. A confined service process, such as the above, would be able to process data in both A_1 and A_2, but it would not be allowed to output information that associated names with salaries. The utility of this, in solving the usually vexing problems of preventing outputs resulting from cross-correlations of files in data banks, is obvious.

3.7.3 Data Bases and Data Banks

Suppose a system (or network of systems) has a large data base containing different classes of information about individuals. For example, one class may contain employment records, a second health records, a third credit reports, a fourth criminal records, and so forth forth. We assume that all access to the data base is restricted to a set of procedures P, so that if a process q requires information from the data base, it must issue a query that in turn invokes a procedure $p \in P$. Now, suppose that associated with q is a clearance level, $\underline{q}$, meaning that q is not permitted access to information in class A unless $A \rightarrow \underline{q}$ (see Section 2.4.1). To guarantee the security of the system, it is therefore necessary to prove that no response r returned to q is derived from information in classes $A_1, \ldots, A_n$ unless $A_1 \oplus \ldots \oplus A_n \rightarrow \underline{q}$. Clearly, this property is guaranteed to hold as long as each $p \in P$ is certified secure. The significance of this result must not be overlooked — it says that we can guarantee the security of the system.

We do not claim our mechanism is sufficient to handle all of the security requirements of a data base system. For example, it does not appear to be applicable to certain "data dependent" access requirements where access to a datum (e.g., the SALARY field of an employee's record) may depend on the value of the datum (e.g., SALARY < 10000). See [CH75,CM72,Hf71,Ts74] for a general discussion of the security requirements of data base systems. Future research is needed to determine the applicability of our mechanism to these systems.

PROGRAM CERTIFICATION UNDER DYNAMIC BINDING

Chapter 4

In Section 2.4.5 we examined the security requirements of a multiclass system with dynamic binding of objects to security classes. We observed that this type of system requires both a run-time mechanism that dynamically updates security classes and a compile-time mechanism that inserts special code into a program (so that all implicit flows are accounted for) and certifies the secure execution of the transformed program. The purpose of this chapter is to describe these mechanisms.

As in the previous chapter, we restrict attention to highly structured programs (even for which, as will be seen, certification is not easy). Define a program (or statement) Φ recursively by:

1. Φ is an assignment statement "b := $f(a_1,\ldots,a_n)$", where $f(a_1,\ldots,a_n)$ denotes an expression with operands $a_1,\ldots,a_n$.
2. Φ is a compound statement (or concatenation) "begin $\Phi_1;\ldots;\Phi_m$ end", where $\Phi_1,\ldots,\Phi_m$ are programs.
3. Φ is a selection statement "if e then Φ_1 [else Φ_2]", where e is a Boolean variable and Φ_1 and Φ_2 are programs.
4. Φ is an iteration statement "while e do Φ_1", where e is a Boolean variable and Φ_1 is a program.

No generality is lost by restricting the "e" in selection and iteration statements to a Boolean variable, since "if ξ then Φ_1 else Φ_2"

(where ξ is an expression) is equivalent to "begin e := ξ; if e then Φ_1 else Φ_2 end" and "while ξ do Φ_1" is equivalent to "begin e := ξ; while e do begin Φ_1; e := ξ end end". Likewise, no generality is lost by excluding a case statement, for statement, or repeat-until statement: a case statement can be transformed into a sequence of nested if-then-else statements; for statements and repeat-until statements can be transformed into while-do statements. (For example, the statement "repeat Φ_1 until e" is equivalent to "begin Φ_1; while e do Φ_1 end".)

A given statement Φ of a program is said to be at nesting level k ($k \geq 0$) if it is textually enclosed by k selection or iteration statements. Note that if a program Φ is at nesting level k, then execution of Φ is conditioned on the values of exactly k Boolean variables, and all implicit flows to variables assigned values in Φ occur from these k Boolean variables.

The hardware support is described in Section 4.1. It consists of tags for dynamically binding objects to security classes and a stack for storing the classes of Boolean variables. Each register of the machine contains a tag field that holds the class of the information stored in it. Immediately prior to execution of a statement Φ at nesting level k, the stack contains the classes of the k Boolean variables on which the execution of Φ is conditioned. A special hardware mechanism updates the tag field of a register receiving a result, using the tag field of the input operands (explicit flows) and the classes on the stack (implicit flows) to derive the new tag setting.

The software support is described in Section 4.2. It consists of a compile-time procedure (called "Algorithm T1") that transforms a given program into a secure sequence of instructions for the hardware. This is done by inserting update instructions into the program. These instructions cause the tag field of a register receiving an implicit flow from a Boolean variable to be updated even if no explicit assignment is made to the register. Algorithm T1 is easily incorporated into the analysis phase of a compiler and does not substantially increase the compilation time of a program. However, it may substantially increase its execution time (because of the update instructions).

The run-time inefficiencies wrought by Algorithm T1 lead us in Section 4.3 to investigate a more sophisticated transformation procedure ("Algorithm T2") that analyzes the flow of information through a program and generates update instructions only where security cannot otherwise be guaranteed. This works as follows. Beginning with assumptions about the initial classes of the input parameters of the program, Algorithm T2 simulates information flow as it will occur during program execution until it is able to determine upper and lower bounds on the class of each variable after each statement is executed. It then inserts an update instruction into any statement which can cause an implicit flow from a Boolean variable e to a variable b and the bounds on $\underline{e}$ and $\underline{b}$ are such that $\underline{e} \rightarrow \underline{b}$ is not otherwise guaranteed to hold. In this case, run-time support is necessary to verify the correctness of the initial classes of the input variables when the program is entered. Algorithm T2 has the advantage of adding a smaller factor to the time and space requirements of a program (indeed, it may not be necessary to generate any update instructions at all) but the disadvantage of being considerably more complex.

In Section 4.4 we consider possible extensions to the mechanism for supporting files, arrays, procedures, interrupts, static binding, and a proposed *verify* statement. Files and arrays are easily handled with minor modifications to the hardware and software mechanisms. Procedures are also easily handled provided they do not reference global variables. Interrupts are more difficult, however, since the possibility of an interrupt must be considered during the transformation of a program. Support for objects that are statically bound is easily added, so that the mechanism is applicable to programs which reference both statically bound and dynamically bound variables. The proposed *verify* statement is a language extension allowing greater flexibility in the specification of the initial classes of the input parameters (for Algorithm T2). Although useful, this feature greatly complicates the implementation of T2. Further investigation is needed to determine its feasibility.

4.1 Hardware Mechanism

Under dynamic binding we need an efficient mechanism for obtaining and updating the security class of an object. This objective is achieved with a tagged architecture [Fu73] that includes tags, a stack, and a class updating mechanism.

With each register u there is associated a tag field whose contents identify the security class bound to u — i.e., the value of the function *tag*(u) described in Section 2.1. For an object a stored in location *loc*(a), we shall employ the usual notation "$\underline{a}$" to denote the tag field of *loc*(a) and simply "a" to denote its value field. Tags are associated with both arithmetic and storage registers.

A hardware stack HS is used for saving and restoring the classes of Boolean variables. As noted earlier, a statement Φ at nesting depth k is conditioned on the values of k Boolean variables $e_1,\ldots,e_k$ (unless $k = 0$), and information flows from these variables to any variable b assigned a value in Φ. Rather than storing the classes $\underline{e}_1,\ldots,\underline{e}_k$ separately in HS, it is convenient to initialize HS with $\underline{e}_0 = L$ (the lowest class) and place $\underline{e}_0 \oplus \ldots \oplus \underline{e}_i$ at stack position i ($0 \leq i \leq k$), so that the top of the stack is the least upper bound of $\{\underline{e}_1,\ldots,\underline{e}_k\}$. This is done as follows. Denote by $\underline{HS}$ the class on the top of HS, and let push(e) be an operation which computes $\underline{HS} \oplus \underline{e}$ and adds this to the stack. (To an observer, this has the effect of setting $\underline{HS} := \underline{HS} \oplus \underline{e}$.) After $e_1,\ldots,e_k$ have been pushed onto HS, $\underline{HS} = \underline{e}_1 \oplus \ldots \oplus \underline{e}_k$. Is is then possible to update the class of the variable b to reflect the implicit flows from $e_1,\ldots,e_k$ to b by performing the single operation $\underline{b} := \underline{b} \oplus \underline{HS}$. The operation pop removes and discards the value on top of HS. (To an observer, this has the effect of restoring $\underline{HS}$ to the value it had prior to the corresponding push operation.)

The push and pop operations are to be performed when (and only when) a selection or iteration statement is executed as follows:

Statement	Execution
if e then Φ_1 [else Φ_2]	push(e); if e then Φ_1 [else Φ_2]; pop
while e do Φ_1	push(e); while e do begin Φ_1; pop; push(e) end pop

For the present we leave unspecified how the push and pop operations are associated with these statements; in the next section we shall show how this can be done with a software mechanism.

Assume $\underline{HS} = A$ immediately prior to the execution of a program Φ. It is easily verified from the specifications above and the definition of a program that the following propositions hold:

Proposition 4.1-1. $\underline{HS} = A$ immediately after execution of Φ (since selection and iteration statements both restore $\underline{HS}$ on termination).

Proposition 4.1-2. $A \rightarrow \underline{HS}$ during execution of Φ (since the operation push(e) sets $\underline{HS} := \underline{HS} \oplus \underline{e}$).

Proposition 4.1-3. If $\Phi =$ "begin $\Phi_1;\ldots;\Phi_m$ end", then $\underline{HS} = A$ immediately before and after execution of each Φ_i $(1 \leq i \leq m)$.

Proposition 4.1-4. If $\Phi =$ "if e then Φ_1 [else Φ_2]", then $\underline{HS} = A \oplus \underline{e}$ immediately before and after execution of Φ_1 [or Φ_2]. Furthermore, since $\underline{e} \rightarrow A \oplus \underline{e}$, $\underline{e} \rightarrow \underline{HS}$ during execution of Φ_1 [or Φ_2].

Proposition 4.1-5. If $\Phi =$ "while e do Φ_1", then $\underline{HS} = A \oplus \underline{e}^i$ immediately before and after execution of Φ_1 on the i^{th} iteration, where $\underline{e}^i$ is the class of e on the i^{th} iteration. Furthermore, since $\underline{e}^i \rightarrow A \oplus \underline{e}^i$, $\underline{e}^i \rightarrow \underline{HS}$ during execution of Φ_1. Since $A \oplus \underline{e}^i$ is popped from HS at the end of the i^{th} iteration, the size of HS is always $k + 2$ (where k is the nesting depth of Φ) immediately before and after execution of Φ_1.

Proposition 4.1-6. $A = L$ if Φ is at nesting level 0.

The class updating mechanism automatically updates the class of an object b when an assignment statement $\Phi =$ "$b := f(a_1,\ldots,a_n)$" is executed, using the classes of the operands and the top of the stack. Specifically, it sets $\underline{b} := \underline{a_1} \oplus \ldots \oplus \underline{a_n} \oplus \underline{HS}$.

As an illustration of how this can be done, suppose the statement "a := b * c + d" is translated into the following code for a single accummulator machine:

```
LOAD b
MULT c
ADD  d
STO  a
```

Letting ACC denote the accummulator, the hardware would be designed to perform the following operations for each instruction:

Instruction	Operation on Data	Operation on Tags
LOAD b	ACC := b	$\underline{ACC} := \underline{b} \oplus \underline{HS}$
MULT c	ACC := ACC * c	$\underline{ACC} := \underline{ACC} \oplus \underline{c} \oplus \underline{HS}$
ADD d	ACC := ACC + d	$\underline{ACC} := \underline{ACC} \oplus \underline{d} \oplus \underline{HS}$
STO a	a := ACC	$\underline{a} := \underline{ACC} \oplus \underline{HS}$

It is especially important to note that the class $\underline{HS}$ is used at every step to update tags. This insures that implicit flows are properly accounted for when new values are produced.

Given an assignment statement Φ = "b := f($a_1,\ldots,a_n$)", the update operation "$\underline{b} := \underline{a}_1 \oplus \ldots \oplus \underline{a}_n \oplus \underline{HS}$" clearly has the effect of setting $\underline{b}$ to be at least as great as the classes of all variables that explicitly flow into b. That it also sets $\underline{b}$ to be at least as great as the classes of all variables that implicitly flow into b has been suggested earlier and will be established rigorously below by Theorem 4.1-1, which states that $\underline{HS}$ is the least upper bound of the classes of all values on which Φ is conditioned. To prove the theorem, we need the result of the following lemma which states that for an iteration "while e do Φ_1", $\underline{e}$ can only increase from one iteration to the next.

Lemma 4.1-1. Consider the execution of an iteration statement Φ = "while e do Φ_1". If $\underline{e}^i$ is the class of e immediately prior to the i^{th} iteration of Φ_1 $(i \geq 1)$, then $\underline{e}^{i-1} \rightarrow \underline{e}^i$ $(i > 1)$.

proof. By Proposition 4.1-5, $\underline{e}^{i-1} \rightarrow \underline{HS}$ during the $(i-1)^{st}$ iteration of Φ_1. We then have

$$\underline{e}^i = \begin{cases} \underline{e}^{i-1} & \text{if e is not assigned a value on the } (i-1)^{st} \text{ iteration,} \\ \underline{a_1} \oplus \ldots \oplus \underline{a_n} \oplus A & \text{if the last assignment to e on the } (i-1)^{st} \text{ iteration associated with e some value } f(a_1,\ldots,a_n) \text{ when } \underline{HS} = A. \end{cases}$$

But $\underline{e}^{i-1} \rightarrow A$, so that $\underline{e}^{i-1} \rightarrow \underline{e}^i$.

Theorem 4.1-1. Consider the execution of a program, and let Φ be the next statement to be executed. Let $E(\Phi)$ be the set of values on which the execution of Φ is conditioned. Then $\underline{HS} = \underline{E(\Phi)}$ immediately before and after execution of Φ, where

$$\underline{E(\Phi)} = \begin{cases} \{\underline{e} \mid e \in E(\Phi)\} & \text{if } E(\Phi) \neq \emptyset \\ L & \text{otherwise} \end{cases}$$

proof. If Φ is a member of a sequence "begin $\Phi_1;\ldots;\Phi_m$ end" (where m = 1 implies that begin and end need not be present), $E(\Phi) = E(\Phi_1)$; and by Proposition 4.1-3, $\underline{HS} = \underline{E(\Phi)}$ prior to executing Φ if and only if $\underline{HS} = \underline{E(\Phi_1)}$ prior to executing Φ_1. Therefore we may assume without loss of generality that Φ is the first member of a sequence.

Now, if the nesting level k of Φ is 0, the initial conditions imply $E(\Phi) = \emptyset$ and $\underline{HS} = L$ (by Proposition 4.1-6), so that the theorem holds. As an induction hypothesis, assume the theorem holds for Φ at nesting levels $k < K$ and consider Φ at level K.

There are two possibilities. First, suppose there is a statement $\Phi' =$ "$\underline{\text{if}}$ e $\underline{\text{then}}$ Φ_1 [$\underline{\text{else}}$ Φ_2], where Φ is the first member of sequence Φ_1 [or Φ_2]. Clearly $E(\Phi) = E(\Phi') \cup \{e\}$. Since Φ' is at nesting level K-1, the induction hypothesis implies $\underline{HS} = \underline{E(\Phi')}$ prior to executing Φ'. Since $\underline{HS} \oplus \underline{e}$ is pushed onto the stack HS before executing Φ_1 [or Φ_2], we have

$$\underline{E(\Phi)} = \underline{E(\Phi') \cup \{e\}} = \underline{E(\Phi')} \oplus \underline{e} = \underline{HS} \oplus \underline{e}$$

as required. Second, suppose there is a statement $\Phi' =$ "$\underline{\text{while}}$ e $\underline{\text{do}}$ Φ_1", where Φ is the first member of sequence Φ_1. At the r^{th} iteration,

$$E(\Phi) = E(\Phi') \cup \{e^1,\ldots,e^i,\ldots,e^r\}$$

(since execution of each iteration is conditioned on the preceding), where e^i is the value of e prior to the i^{th} iteration. Whence,

$$\underline{E(\Phi)} = \underline{E(\Phi')} \oplus \underline{e}^1 \oplus \ldots \oplus \underline{e}^r.$$

By Lemma 4.1-1, $\underline{e}^{i-1} \rightarrow \underline{e}^i$ $(1 < i \leq r)$ so that $\underline{e}^1 \oplus \ldots \oplus \underline{e}^r = \underline{e}^r$. By Proposition 4.1-5, $\underline{HS} \oplus \underline{e}^r$ will be on the stack just prior to the r^{th} iteration; and by induction $\underline{HS} = \underline{E(\Phi')}$, so that

$$\underline{E(\Phi)} = \underline{E(\Phi')} \oplus \underline{e}^r = \underline{HS} \oplus \underline{e}^r$$

as required.

Figure 4.1-1 illustrates the operation of the hardware mechanism by tracing the execution of the insecure program cited in Section 2.4.5. The two traces shown illustrate the claim made earlier, that b = a on termination of the program , while $\underline{b} = L$ even though $\underline{a} = H$ (hence, $\underline{a} \not\rightarrow \underline{b}$ as required). This program demonstrates that the hardware mechanism is insufficient to guarantee security, and that a software mechanism is required.

Source Statement	Execution Trace	Stack (HS at right)	a,a	b,b	c,c	e,e
initial conditions for the case a = 0:		L	H,0	L,0	L,0	L,false
begin						
e := (a = 0);	e := (a = 0)					H,true
if e then c := 1;	push(e)	L, H				
	c := 1				H,1	
	pop	L				
e := (c = 0);	e := (c = 0)					H,false
if e then b := 1	push(e)	L,H				
	pop					
end						
final conditions (b = a = 0):		L	H,0	L,0	H,1	H,false
initial conditions for the case a = 1:		L	H,1	L,0	L,0	L,false
begin						
e := (a = 0);	e := (a = 0)					H,false
if e then c := 1;	push(e)	L, H				
	pop	L				
e := (c = 0);	e := (c = 0)					L,true
if e then b := 1	push(e)	L,L				
	b := 1			L,1		
	pop	L				
end						
final conditions (b = a = 1):		L	H,1	L,1	L,0	L,true

Figure 4.1-1. Execution Traces of an Insecure Program.

4.2 Software Mechanism

The hardware mechanism outlined in the previous section is not sufficient to guarantee security since an implicit flow from a Boolean variable e to a variable b can occur in the absence of any explicit flow to b ($\underline{b}$ is updated only when an explicit assignment is made, but e flows implicitly to b even when the statements assigning values to b are not executed). In general, this may occur during execution of a selection statement, "if e then Φ_1 [else Φ_2]", if there is an assignment statement "b := $f(a_1,\ldots,a_n)$" in Φ_1 [or Φ_2] and that statement is not executed. This may also occur during execution of an iteration statement, "while e do Φ_1", if there is an assignment statement "b := $f(a_1,\ldots,a_n)$" in Φ_1, since no statement in Φ_1 is executed after the final evaluation of e (which necessarily leaves e = false).

This problem can be solved with a compile-time mechanism that inserts into the compiled code an operation "update b", having the effect

$$\underline{b} := \underline{b} \oplus \underline{HS},$$

at all locations where an assignment to b is conditioned on one or more Boolean variables $e_1,\ldots,e_k$ (recall from Theorem 4.1-1 that $\underline{HS}$ = $\underline{e}_1 \oplus \ldots \oplus \underline{e}_k$ at that time). Then all implicit flow to b is guaranteed secure irrespective of whether or not the assignment statement is executed. The mechanism also inserts into the compiled code all necessary push and pop instructions.

Rather than show the details of the compilation process, we have expressed the mechanism as a high level recursive procedure T1(Φ, V) which transforms an arbitrary program Φ into an equivalent secure one by inserting all necessary push, pop, and update instructions (see Figure 4.2-1). The parameter V returns the set of variables that appear on the left side of assignment statements in Φ (i.e., V is equivalent to the set V(Φ) introduced in Section 2.4.4 and referenced in Chapter 3); it is used to generate update instructions. (We have written update X to denote a sequence of update instructions, one for each member of the set X.) The language used to express T1 resembles PASCAL [Wr71] and includes the types "program" and "variable"; their use should be clear in Figure 4.2-1. The initial call to T1 is T1(Φ,V) where Φ is an entire program. T1 returns the transformed (secure) program in Φ and the set of variables that appear on the left side of assignment statements in V. Update instructions are generated as needed for elements of V (the proof that this is done correctly is given below).

Figure 4.2-2 (a) shows the results of the transformation when T1 is applied to the sample program of Figure 4.1-1. The reader may easily verify that the update instructions cause b to be identical to a on termination of the program, whereupon the "leak" has been plugged. Figure 4.2-2 (b) shows an equivalent program, where the same leak can occur using iteration statements; in this case also, the transformed program is secure.

Theorem 4.2-1. Let Φ be a given program. After execution of a T1(Φ,V) the following will be true:

1) V will be the set of all variables appearing on the left side of assignment statements in Φ; and

```
T1: procedure ($\Phi$: program, V: powerset variable);

  $V_1$, $V_2$: powerset variable;
  i: integer;

  do case $\Phi$ of

    $\Phi$ = "b := $f(a_1,\ldots,a_n)$":          {Assignment}
      V := b;

    $\Phi$ = "begin $\Phi_1;\ldots;\Phi_m$ end":     {Concatenation}
      begin
        V := $\emptyset$;
        for i := 1 to m do begin
          T1($\Phi_i$,$V_1$);
          V := $V \cup V_1$  end
      end;

    $\Phi$ = "if e then $\Phi_1$ [else $\Phi_2$]":  {Selection}
      begin
        T1($\Phi_1$,$V_1$);
        if $\Phi_2 \neq \emptyset$ then T1($\Phi_2$,$V_2$) else $V_2$ := $\emptyset$;
        V := $V_1 \cup V_2$;
        $\Phi$ := "begin  push(e); if e then $\Phi_1$ [else $\Phi_2$];
                 update $V - V_1 \cap V_2$; pop end"
      end;

    $\Phi$ = "while e do $\Phi_1$":              {Iteration}
      begin
        T1($\Phi_1$,V);
        $\Phi$ := "begin push(e); while e do begin $\Phi_1$; pop; push(e) end
                 update V; pop end"
      end

    endcase

end T1
```

Figure 4.2-1. Transformation Algorithm T1.

```
    Insecure Source Programs        Transformed Secure Program

a)  begin                           begin
      b := 0;                         b := 0;
      c := 0;                         c := 0
      e := (a = 0);                   e := (a = 0);
      if e then c := 1;               begin push(e); if e then c := 1;
                                        update c; pop end;
      e := (c = 0);                   e := (c = 0);
      if e then b := 1                begin push(e); if e then b := 1;
                                        update b; pop end
    end                             end


b)  begin                           begin
      b := 0;                         b := 0;
      c := 0;                         c := 0;
      e := (a = 0);                   e := (a = 0);
      while e do begin                begin push(e);
        c := 1; e := false end          while e do begin begin c := 1;
                                          e := false end; pop; push(e) end;
                                        update c, e; pop end;
      e := (c = 0);                   e := (c = 0);
      while e do begin                begin push(e);
        b := 1; e := false end          while e do begin begin b := 1;
                                          e := false end; pop; push(e) end;
                                        update b, e; pop end
    end                             end
```

Figure 4.2-2. Transformation of Insecure Programs.

2) Φ will be secure. Specifically, all necessary <u>push</u>, <u>pop</u>, and <u>update</u> instructions will have been inserted into Φ so that after any execution of Φ, $\underline{HS} \rightarrow \underline{b}$ will hold for all variables $b \in V$. Therefore, if Φ is conditioned on Boolean variables $e_1, \ldots, e_k$, $\underline{e_i} \rightarrow \underline{b}$ $(1 \leq i \leq k)$ is guaranteed to hold for all $b \in V$ (since $\underline{HS} = \underline{e_1} \oplus \ldots \oplus \underline{e_k}$ on entry to Φ by Theorem 4.1-1). Moreover, if Φ is a selection or iteration statement with Boolean variable e, $\underline{HS} \oplus \underline{e} \rightarrow \underline{b}$ will hold for all $b \in V$.

<u>proof</u>. Define the "structure index" d of a program to be the number of concatenation, selection, and iteration structure operations used to construct it. We proceed by induction on the value d of Φ.

BASIS (d=0): In this case Φ is an assignment "$b := f(a_1, \ldots, a_n)$". On termination, T1 leaves Φ unchanged and $V = \{b\}$. Since the hardware updating mechanism guarantees that $\underline{HS} \rightarrow \underline{b}$ after the assignment is performed, T1 correctly processes Φ.

INDUCTION: Let Φ have structure index $d = D$ and suppose the theorem holds for $d < D$. There are three cases.

i) Φ = "<u>begin</u> $\Phi_1; \ldots ; \Phi_m$ <u>end</u>". In this case, T1 may be assumed to correctly process Φ_i $(1 \leq i \leq m)$, since Φ_i has structure index $d_i < D$. By Proposition 4.1-3, if $\underline{HS} = A$ prior to execution of Φ, then $\underline{HS} = A$ prior to execution of each Φ_i. Therefore, for $b \in V_i$ (where V_i is just the set of variables assigned values in Φ_i), $\underline{HS} \rightarrow \underline{b}$ will hold after execution of Φ_i. Hence, for $b \in V_1 \cup \ldots \cup V_m$, $\underline{HS} \rightarrow \underline{b}$ will hold after execution of Φ. Since T1 returns $V = V_1 \cup \ldots \cup V_m$ and no push, pop, or update instructions need be inserted into Φ, T1 correctly processes Φ.

ii) Φ = "if e then Φ_1 [else Φ_2]". In this case, T1 may be assumed to correctly process Φ_1 [and Φ_2] since Φ_1 and Φ_2 have $d_1 < D$ and $d_2 < D$. Therefore $V = V_1 \cup V_2$. The transformed program is "begin push(e); if e then Φ_1 [else Φ_2]; update $V - V_1 \cap V_2$; pop end". Therefore, if Φ_1 is executed, then for $b \in V_1$, $\underline{HS} \oplus \underline{e} \rightarrow \underline{b}$ will hold after execution of Φ_1; if Φ_2 is executed, then for $b \in V_2$, $\underline{HS} \oplus \underline{b} \rightarrow \underline{b}$ will hold after execution of V_2. Hence, the update instruction guarantees that for $b \in V$, $\underline{HS} \rightarrow \underline{HS} \oplus \underline{e} \rightarrow \underline{b}$ will hold after execution of Φ, irrespective of whether or not Φ_1 [or Φ_2] is executed.

iii) Φ = "while e do Φ_1". In this case, T1 may be assumed to correctly process Φ_1 since $d_1 < D$. Therefore $V = V_1$. The transformed program is "begin push(e); while e do begin Φ_1; pop; push(e) end; update V; pop end". In this case it is trivially true that for $b \in V$, $\underline{HS} \rightarrow \underline{HS} \oplus \underline{e} \rightarrow \underline{b}$ will hold after execution of Φ, since the class of each $b \in V$ is updated by the update instruction.

The reader may have wondered if insertion of push and pop instructions is necessary. If the hardware is designed to automatically push the stack on execution of any conditional branch, there will be no need for the compiler to insert push instructions into the compiled code. To see this, consider the translation of selection and iteration statements. Assume that Boolean false is represented internally by 0 and true by any non-zero value. Let B be an unconditional branch instruction and BZ a branch on zero instruction. The translation of selection and iteration statements into code for a machine with a single accummulator (ACC) is:

a) begin push(e); if e then Φ_1 else Φ_2; update $V - V_1 \cap V_2$; pop end

```
     LOAD e
     PUSH
     BZ   L1
     code for Φ1
     B    L2
L1   code for Φ2
L2   code for update
     POP
```

b) begin push(e); while e do begin Φ_1; pop; push(e) end; update V; pop end

```
L1   LOAD e
     PUSH
     BZ   L2
     code for Φ1
     POP
     B    L1
L2   code for update
     POP
```

The PUSH instruction pushes HS ⊕ ACC onto the stack. Since no other statement types generate conditional branch instructions, it is clear from the above that the PUSH operation can be performed automatically when the BZ instruction is executed. Note, however, that the compiler must still generate code for the POP operation, since there is no instruction analogous to the BZ instruction that is always paired with POP.

Hereafter, we will assume without loss of generality that the operation push(e) is performed automatically and adopt the simpler forms for transformed iteration and selection statements:

```
begin if e then $\Phi_1$ [else $\Phi_2$]; update V - $V_1 \cap V_2$; pop end
begin while e do begin $\Phi_1$; pop end; update V; pop end
```

The mechanism outlined above may substantially increase the time and space requirements of a program. For example, the program

```
if el
  then if e2 then a := 0 else b := 0
  else c := 0
```

would be transformed into

```
begin
  if el
    then begin if e2 then a := 0 else b := 0; update a, b; pop end
    else c := 0
  update a, b, c;
  pop
end
```

Execution of the transformed program results in as many as five update operations being performed, when only one assignment operation is performed. Furthermore, none of the update operations need be performed at all if el = e2 = L (since they will have no effect). The software mechanism introduced in the next section provides a means for eliminating some (or all) of the update operations when they are not required.

4.3 Software Mechanism for Selective Updating

The mechanism of this section improves on the one of the previous section in two ways: it generates fewer update instructions, and it permits the programmer to declare initial classes for input parameters. Again, our approach is to describe an algorithm that transforms a given program into a secure one.

The syntax of PASCAL is convenient for declaring initial classes for input parameters. For example, f: procedure (x: integer of class A, y: integer of class L; z: integer)

specifies initial class of A and L for formal input parameters x and y respectively. A run-time check must verify the classes of the actual parameters. Thus, if f is invoked by an operation "call f(a, b, c)", the run-time mechanism would check that $\underline{a}$ = A and $\underline{b}$ = B before calling f.

The transformation algorithm T2 accepts a program Φ as input, transforms Φ into a secure program, and then returns the set V of variables to which statements of Φ assign values. It also returns tables, denoted ℓ and h, specifying the lowest and highest possible classes for each variable. In particular, if $\ell[a]$ and h[a] are the lowest and highest classes of variable a before T2 processes Φ, then $\ell[a]$ and h[a] will be updated by T2 so that they represent the lowest and highest possible classes of a after an execution of Φ. The initial call on T2 for a program Φ is of the form T2(Φ, V, ℓ, h), where ℓ and h are initialized as follows: if a is an input variable of declared class A, then $\ell[a]$ = h[a] = A; otherwise $\ell[a]$ = L and h[a] = H. Algorithm T2 is given in Figure 4.3-1.

For each call, T2 simulates information flow during possible executions of the program Φ until bounds on the classes of all variables are uniquely determined. This is done by simulating the effect of Φ on the machine's stack and the classes of all variables. A global variable SS ("software stack") is used by T2 to represent the hardware stack HS, but with one difference: SS contains pairs of classes corresponding to the lowest and highest classes that could be pushed onto HS during an

```
T2: procedure ($\Phi$: program; V: powerset variable;
              $\ell$, h: array [variable] of class);

  $V_1$, $V_2$: powerset variable;
  $\ell_1$, $\ell_2$, $h_1$, $h_2$: array [variable] of class;
  i: integer;
  change: boolean;


  UPDATE: procedure (X, Y: powerset variable);

    Y := $\{b \in X \mid \underline{SS}.h \not\to \ell[b]\}$;
    for all $b \in Y$ do begin $\ell[b] := \ell[b] \oplus \underline{SS}.\ell$; $h[b] := h[b] \oplus \underline{SS}.h$ end

  end UPDATE;


  do case $\Phi$ of        {Assignment, Concatenation, Selection, Iteration}

    $\Phi$ = "b := $f(a_1,\ldots,a_n)$":                {Assignment}

      begin
        $\ell[b] := \ell[a_1] \oplus \ldots \oplus \ell[a_n] \oplus \underline{SS}.\ell$;
        $h[b] := h[a_1] \oplus \ldots \oplus h[a_n] \oplus \underline{SS}.h$;
        V := b
      end;

    $\Phi$ = "begin $\Phi_1;\ldots;\Phi_m$ end":             {Concatenation}

      begin
        V := $\emptyset$;
        for i := 1 to m do
          begin
            T2($\Phi_i$, $V_1$, $\ell$, h);
            V := $V \cup V_1$
          end
      end;
```

Figure 4.3-1. Transformation Algorithm T2.

```
$\Phi$ = "if e then $\Phi_1$ [else $\Phi_2$]" or                    {Selection}
    "begin if e then $\Phi_1$ [else $\Phi_2$]; update U; pop end":

  begin
    PUSH(e);
    $\ell_1, \ell_2 := \ell$;  $h_1, h_2 := h$;
    T2($\Phi_1, V_1, \ell_1, h_1$);
    if $\Phi_2 \neq \emptyset$ then T2($\Phi_2, V_2, \ell_2, h_2$) else $V_2 := \emptyset$;
    $V := V_1 \cup V_2$;
    for all $b \in V$ do
      begin
        $\ell[b] := \ell_1[b] \otimes \ell_2[b]$;
        $h[b] := h_1[b] \oplus h_2[b]$
      end;
    UPDATE($V - V_1 \cap V_2$, U);
    POP;
    $\Phi$ = "begin if e then $\Phi_1$ [else $\Phi_2$]; update U; pop end"
  end;


$\Phi$ = "while e do $\Phi_1$" or                                   {Iteration}
    "begin while e do begin $\Phi_1$; pop end; update U; pop end":

  begin
    $\ell_1 := \ell$;  $h_1 := h$;
    repeat
      begin
        PUSH(e); $V := \emptyset$;
        T2($\Phi_1, V, \ell_1, h_1$);
        for all $b \in V$ do
          begin
            if $h[b] \neq h[b] \oplus h_1[b]$ then
              begin $h[b] := h[b] \oplus h_1[b]$; change := true end;
            if $\ell[b] \neq \ell[b] \otimes \ell_1[b]$ then
              begin $\ell[b] := \ell[b] \otimes \ell_1[b]$; change := true end
          end;
        POP; PUSH(e)
      end
    until $\neg$ change;
    UPDATE(V, U);
    POP;
    $\Phi$ := "begin while e do begin $\Phi_1$; pop end; update U; pop end"
  end

endcase

end T2
```

Figure 4.3-1, cont.

execution of Φ. Denoting by $\underline{SS}.\ell$ and $\underline{SS}.h$ the lowest and highest pair on top of SS, the function PUSH(e) pushes $(\underline{SS}.\ell \oplus \ell[e], \underline{SS}.h \oplus h[e])$ onto SS. Corresponding, POP removes the top pair from SS and discards it. The main complication of the simulation process occurs for iteration statements, Φ = "*while* e *do* Φ_1". For such a statement, the value of $\underline{e}$ (and thus $\underline{HS}$) may increase from one iteration to the next. Therefore, a program Φ already transformed by T2 may require reprocessing to guarantee all variables in need of updating have been detected. Such reprocessing will terminate as soon as the ℓ and h tables have the same values before and after processing an iteration statement — that is, as soon as the values of ℓ and h "stabilize".

The details of the operation of T2 are explained fully by the following theorem which establishes its correctness.

Theorem 4.3-1. Let Φ, ℓ, h, and SS be given such that prior to any execution of Φ

$\ell[a] \rightarrow \underline{a} \rightarrow h[a]$ for each variable a of Φ, and

$\underline{SS}.\ell \rightarrow \underline{HS} \rightarrow \underline{SS}.h$.

After execution of a call T2(Φ, V, ℓ, h) the following will be true:

1) V will be the set of all variables appearing on the left side of assignment statements in Φ;

2) Φ will be secure; that is, all necessary *push*, *pop*, and *update* operations will have been inserted into Φ so that after any execution of Φ, $\underline{HS} \rightarrow \underline{b}$ will hold for all b ε V. Therefore, if Φ is conditioned on Boolean variables $e_1,\ldots,e_k$, $\underline{e_i} \rightarrow \underline{b}$ $(1 \leq i \leq k)$ is guaranteed to hold for all b ε V (since $\underline{HS} = \underline{e_1} \oplus \ldots \oplus \underline{e_k}$ on entry to Φ by Theorem 4.1-1). Moreover,

if Φ is a selection or iteration statement with Boolean variable e, $\underline{HS} \oplus \underline{e} \rightarrow \underline{b}$ will hold for all $b \in V$.

3) ℓ and h will be updated by T2 so that after any execution of Φ

$$\ell[a] \rightarrow \underline{a} \rightarrow h[a] \text{ for all variables } a \text{ of } \Phi.$$

<u>proof</u>. The proof is by induction on the "structure index" d of Φ, as in Theorem 4.2-1.

BASIS (d = 0): In this case Φ = "$b := f(a_1,\ldots,a_n)$", and $V = \{b\}$. The hardware guarantees that $\underline{b} = \underline{a_1} \oplus \ldots \oplus \underline{a_n} \oplus \underline{HS}$ after an execution of Φ, so that $\underline{HS} \rightarrow \underline{b}$ is guaranteed to hold. Since $\underline{SS}.\ell \rightarrow \underline{HS} \rightarrow \underline{SS}.h$ and $\ell[a_i] \rightarrow \underline{a_i} \rightarrow h[a_i]$ $(1 \leq i \leq n)$ prior to an execution of Φ, the lattice properties (see Section 2.3) imply that $\ell[b] = \ell[a_1] \oplus \ldots \oplus \ell[a_n] \oplus \underline{SS}.\ell \rightarrow \underline{b} \rightarrow h[a_1] \oplus \ldots \oplus h[a_n] \oplus \underline{SS}.h = h[b]$ after an execution of Φ, establishing the correctness of the ℓ and h tables returned by T2.

INDUCTION: Let Φ have structure index d = D, and assume T2 satisfies (1) - (3) for any algorithm whose level is d < D. There are three possibilities.

i) Φ = "<u>begin</u> $\Phi_1;\ldots;\Phi_m$ <u>end</u>". In this case each Φ_i has $d_i < D$, so we may assume that T2 transforms it correctly. Therefore, if $\ell[a] \rightarrow \underline{a} \rightarrow h[a]$ prior to an execution of Φ_i $(1 \leq i \leq m)$, the call T2(Φ_i, V_i, ℓ, h) properly updates the ℓ and h tables so that $\ell[a] \rightarrow \underline{a} \rightarrow h[a]$ after an execution of Φ_i. Hence, the ℓ and h tables returned by T2 for Φ are correct. Moreover,

$V = V_1 \cup \ldots \cup V_m$ when T2 completes processing Φ. Finally, the induction assumption guarantees that $\underline{HS} \rightarrow \underline{b}$ for $b \in V_i$ ($1 \leq i \leq m$), whereupon $\underline{HS} \rightarrow \underline{b}$ for all $b \in V$ (since HS is not modified by begin-end statements).

ii) Φ = "if e then Φ_1 [else Φ_2]" or

Φ = "begin if e then Φ_1 [else Φ_2]; update U; pop end".

Since Φ_1 and Φ_2 have $d_1 < D$ and $d_2 < D$, the induction assumption implies that ℓ_1 and h_1 contain correct cla bounds after T2 processes Φ_1, and ℓ_2 and h_2 contain correct class bounds after T2 processes Φ_2. Since ℓ_1 (or ℓ_2) and ℓ, and h_1 (or h_2) and h can differ only for $b \in V_1 \cup V_2$,

$$\ell[b] = \ell_1[b] \oplus \ell_2[b] \rightarrow \underline{b} \rightarrow h_1[b] \oplus h_2[b] = h[b]$$

must hold if either Φ_1 or Φ_2 is executed, establishing the correctness of the ℓ and h tables returned by T2. Now, T2 correctly simulates HS, so that $\underline{SS}.h \oplus h[e]$ is a correct upper bound on $\underline{HS} \oplus \underline{e}$ when UPDATE is called. Since UPDATE generates an update instruction for any $b \in V - V_1 \cap V_2$ for which $\underline{SS}.h \oplus h[e] \not\rightarrow \ell[b]$, this guarantees that

$$\underline{HS} \rightarrow \underline{HS} \oplus \underline{e} \rightarrow \underline{SS}.h \oplus h[e] \rightarrow \ell[b] \rightarrow \underline{b}$$

will hold after an execution of Φ. (Note that, if $b \in V_1 \cap V_2$ the induction assumption implies that $\underline{HS} \rightarrow \underline{HS} \oplus \underline{e} \rightarrow \underline{b}$ must hold if either Φ_1 or Φ_2 is executed; therefore update instructions will not be generated in this case, as for algorithm T1). The call on UPDATE also correct the tables ℓ and h to reflect changes that can occur when an update instruction is executed, establishing the correctness of the ℓ and h tables returned by T2.

iii) Φ = "while e $\underline{\text{do}}$ Φ_1" or

Φ = "$\underline{\text{begin}}$ $\underline{\text{while}}$ e $\underline{\text{do}}$ $\underline{\text{begin}}$ Φ_1; $\underline{\text{pop}}$ $\underline{\text{end}}$; $\underline{\text{update}}$ U; $\underline{\text{pop}}$ $\underline{\text{end}}$". Since Φ_1 had $d_1 < D$, the induction assumption implies that the first call $T2(\Phi_1, V_1, \ell_1, h_1)$ correctly processes Φ_1. Therefore, if Φ_1 is executed (or iterated) at least once, the first iteration of Φ_1 is guaranteed secure, and for all $b \in V = V_1$, $\ell_1[b] \rightarrow \underline{b} \rightarrow h_1[b]$ will hold after the execution of Φ_1. Hence, if Φ_1 is iterated exactly 0 or 1 times,

$$\ell[b] \otimes \ell_1[b] \rightarrow \underline{b} \rightarrow h[b] \oplus h_1[b]$$

will hold after the execution of Φ, establishing the correctness of the ℓ and h tables after T2 has processed Φ_1 once. Now, to determine bounds on $\underline{b}$ after $m \geq 0$ iterations of Φ_1, it is necessary to repeat this procedure until $\ell[b]$ and $h[b]$ stabilize; that is, until further calls to T2 yield bounds for ℓ_1 and h_1 that fall between ℓ and h. Clearly this process must terminate after a finite number of steps, since $\ell[b]$ and $h[b]$ cannot change beyond L and H in a finite lattice. After ℓ and h have been stabilized, the top of the stack SS contains the upper bound $\underline{SS}.h \oplus h[e]$, where $h[e]$ is the (stabilized, maximal) class of e. The call on UPDATE then guarantees that $\underline{\text{update}}$ instructions are generated for all $b \in V$ for which $\underline{SS}.h \oplus h[e] \not\rightarrow \ell[b]$, so that

$$\underline{HS} \rightarrow \underline{HS} \oplus \underline{e} \rightarrow \underline{SS}.h \oplus h[e] \rightarrow \ell[b] \rightarrow \underline{b}$$

will hold after an execution of Φ. The call on UPDATE also corrects the ℓ and h tables to account for bound changes caused by the $\underline{\text{update}}$ instructions.

Figure 4.3-2 illustrates the result of applying T1 to a sample program, and Figure 4.3-3 illustrates T2 applied to the same program. In Figure 4.3-3 we have traced the simulation process showing the changes to the outermost ℓ and h tables and the stack SS. We assume the classes A, B, and C are linearly related by $A \to B \to C$. Note that the body of the while statement requires two applications of T2 for stabilization. T2 produces three less update instructions than T1 for this program.

The proof of Theorem 4.3-1 suggests that the execution time of T2 can be considerable in comparison with that of T1. The following theorem gives a bound on the most time consuming part of T2.

Theorem 4.3-2. Let Φ be an iteration "while e do Φ_1". Then $T2(\Phi, V, \ell, h)$ will complete after at most $(\delta+1)^{|V|}$ simulations of Φ_1, where δ is the maximum path length (i.e., depth) in the precedence graph of the security class lattice, and $|V|$ is the number of elements in V.

proof. As noted in the proof of Theorem 4.3-1, each simulation of Φ_1 can alter $\ell[b]$ or $h[b]$ for $b \in V$ only. Let $\ell^m[b]$ and $h^m[b]$ denote the value of $\ell[b]$ and $h[b]$ after the mth iteration. Since $\ell^m[b] = \ell^{m-1}[b] \otimes \ell_1[b]$ and $h^m[b] = h^{m-1}[b] \oplus h_1[b]$, we have

$$L \to \ell^m[b] \to \ell^{m-1}[b] \to \ldots \to \ell^1[b] \to$$
$$h^1[b] \to \ldots \to h^{m-1}[b] \to h^m[b] \to H.$$

From this it is clear that for each $b \in V$, $\ell[b]$ and $h[b]$ together can assume at most δ distinct classes, and $\ell[b] = h[b]$ is possible for at most one class. Hence $m \leq (\delta+1)^{|V|}$.

Source Program

```
begin
  e:=b>0;
  while e do
    begin c:=c+1; b:=b-c; e:=b>0 end;






  e:=a>0;
  el:=b<0;
  if e
    then begin if el then c:=0;
      b:=l end
    else c:=l



end
```

Transformed Program

```
begin
  e:= b>0;
  begin
    while e do
      begin
        begin c:=c+l; b:=b-c; e:=b>0 end;
        pop
      end;
    update b,c,e; pop
  end;
  e :=a>0;
  el:=b<0;
  begin
    if e
      then begin begin if el then c:=0; update c;pop end;
        b:=l end
      else c:=l;
    update b; pop
  end
end
```

Figure 4.3-2. Example of a Program Transformed by Tl.

Source Program	Simulation	SS (SS at right)	ℓ,h values at outer level					Transformed Program
			a	b	c	e	el	
initial conditions:		(L,L)	A,A	B,B	C,C	L,H	L,H	
begin								begin
e:=t>0;	e:=b>0					B,B		e:=b>0;
while e do	PUSH(e)	(L,L),(B,B)						begin
begin								while e do
c:=c+1;	c:=c+1							begin
b:=b-c;	b:=b-c							begin c:=c+1; b:=b-c;
e:=b>0	e:=b 0							e:=b>0 end;
end;	POP	(L,L)		B,C		B,C		pop
	PUSH(e)	(L,L),(B,C)						end;
	c:=c+1							update b,e; pop
	b:=b-c							end;
	e:=b>0							
	UPDATE							
	POP	(L,L)						
e:=a>0;	e:=a>0					A,A		e:=a>0;
el:=b<0;	el:=b<0						B,C	el:=b<0;
if e	PUSH(e)	(L,L),(A,A)						begin
then begin								if e
if el then	PUSH(el)	(L,L),(A,A),(B,C)						then begin begin
c:=0;	c:=0							if el then c:=0;pop
	UPDATE							end; b:=1 end
	POP	(L,L),(A,A)						else c:=1;
b:=1 end	b:=1							pop end
else c:=1	c:=1							end
	UPDATE							
	POP	(L,L)						
end								end

Figure 4.3-3. Example of a Program Transformed by T2.

Therefore, we shall make no claims to the effect that T2 is efficient! At this stage in our research, we are primarily interested in the feasibility of performing such a transformation, and so we have sought a representation relatively easy to understand and prove correct. Making an implementation of T2 practical is a matter for future study. An investigation of the techniques used by compilers to analyze the flow of data in a program for the purpose of code optimization (see, for example, [Al70,Gi71]) may suggest some approaches.

It is interesting that despite the complexity of T2, it cannot be guaranteed to generate the minimal number of update instructions; that is, it may insert update instructions into a program that have no effect when executed, independent of the values of the input parameters. This could result, for example, by simulating the body of a while statement more times than it could ever execute, and thereby arriving at ℓ and h bounds on a variable that are either too high or too low. Another way in which this could result is illustrated by the following program:

```
begin
  b := a;
  e := (a = 0);
  if e then b := 1
end,
```

where $\ell[a] = L$, $h[a] = H$, and $SS.h = SS.\ell = L$ on entry to T2. After T2 processes the first two assignment statements, $\ell[b] = \ell[e] = L$ and $h[b] = h[e] = H$. Therefore, when T2 processes the if statement, it pushes (L,H) onto SS. Since $SS.h \neq \ell[b]$, it will generate an instruction "update b" to be executed on completion of the if statement. However, this is unnecessary since b will be the same as e, and therefore HS, for any execution of the program. This problem may be solved with a more complicated mechanism that keeps additional information on the classes of the variables, but we leave this for future study. Here again, the techniques used by optimizing compilers to analyze data flow may prove applicable to this problem.

4.4 Extensions

4.4.1 Files and I/O

Files and I/O operations are easily handled with minor extensions to the basic hardware and software mechanisms. Within the tagged architecture, hardware tags can be associated with files (e.g., as part of a file descriptor [Fu73]) just as with scalar variables. For a file a, we let $\underline{a}$ denote the tag associated with a.

For an *input* statement:

input $b_1, \ldots, b_n$ *from* a,

the class updating mechanism sets

$\underline{b_i} := \underline{a} \oplus \underline{HS} \quad (1 \leq i \leq n)$.

For an output statement

output $a_1, \ldots, a_n$ *to* b,

the class updating mechanism sets

$\underline{b} := \underline{b} \oplus \underline{a_1} \oplus \ldots \oplus \underline{a_n} \oplus \underline{HS}$.

Note that in the case of *output* the prior class of b derives the new class of b, because the contents of b are extended (rather than replaced) to include $a_1, \ldots, a_n$. The requisite modifications to the certification algorithms T1 and T2 are straightforward. For example, *update* instructions must be generated for files as for scalar variables.

4.4.2 Arrays

Although it is possible under dynamic binding to permit the elements of an array to be bound to different classes, this approach can be rejected for at least two reasons. First, the usual semantic definition of an array in programming languages requires all elements to be of the same data type — e.g., in PASCAL

a: array [index] of type.

The same semantic reasons motivating this can be used to motivate uniformity of security classes within an array. Second, the implementation of T2 would become hopelessly complex, for the ℓ and h tables would have to contain entries for each array element. We therefore require arrays to be bound to a single class and assume that the tag of the array is a component of the array descriptor. As usual, the value of the tag for array a is denoted $\underline{a}$.

Associate with array a the class variable $\underline{ra}$, which is set to contain the class of an array reference $a[i_1,\ldots,i_n]$; corresponding to any such reference,

$$\underline{ra} := \underline{a} \oplus \underline{i_1} \oplus \ldots \oplus \underline{i_n}.$$

The variable $\underline{ra}$ might be stored in the array descriptor and updated automatically by hardware when an array element is addressed [Fu73]. Whenever $a[i_1,\ldots,i_n]$ appears on the left side of an assignment statement, the hardware performs the operation

$$\underline{a} := \underline{a} \oplus \underline{ra} \oplus \underline{HS}.$$

As in the case of files, $\underline{a}$ is updated from a prior value of $\underline{a}$, since the array is being modified rather than replaced.

Extensions to T2 are simple in this case. Array references must be treated as outlined above.

4.4.3 Procedure Calls

Let $f(x_1,\ldots,x_m;y_1,\ldots,y_n)$ be a procedure with formal input parameters $x_1,\ldots,x_m$ and formal output parameters $y_1,\ldots,y_n$. Consider a statement

$\underline{\text{call}}\ f(a_1,\ldots,a_m;b_1,\ldots,b_n)$,

where $a_1,\ldots,a_m$ are the actual input parameters and $b_1,\ldots,b_n$ are the actual output parameters. On entry to f, the run-time mechanism sets

$\underline{x}_i := \underline{a}_i \oplus \underline{HS} \quad (1 \leq i \leq m)$

and verifies that $\underline{a}_i = A_i$ if x_i was declared to have initial class A_i in procedure f. On exit, the run-time mechanism sets

$\underline{b}_j := \underline{y}_j \oplus \underline{HS} \quad (1 \leq j \leq n)$.

As usual, if the call statement is conditioned on the value of one or more Boolean variables, it may be necessary to update some of the $\underline{b}_j$ by $\underline{HS}$ on completion of the conditional statement(s). As in Section 3.6, for the moment we assume that procedure f does not write into objects that are global or have a "lifetime" exceeding the activation of f.

Although the modifications to T1 to allow procedure calls are straightforward, those for T2 are more complex if T2 does not have bounds for $\underline{y}_1,\ldots,\underline{y}_n$ when the call statement is processed (since it is then unable to determine the upper bounds for $\underline{b}_1,\ldots,\underline{b}_n$). The difficulty is resolved in the restricted case where the output parameters are derived entirely from the input parameters and data belonging to the lowest class L. In this case, the bounds on $\underline{b}_1,\ldots,\underline{b}_n$ are determined by

$\ell[b_j] := \underline{SS}.\ell$, and

$h[b_j] := h[a_1] \oplus \ldots \oplus h[a_m] \oplus \underline{SS}.h \quad (1 \leq j \leq n)$.

Consider now the case where f writes into objects $z_1,\ldots,z_r$ that are global or have a "lifetime" exceeding the activation of f. There are two reasons why our present mechanism is inadequate to deal with this case. First, if f writes into such objects and the call to f is conditioned on Boolean variables $e_1,\ldots,e_k$, then $\underline{z}_1,\ldots,\underline{z}_r$ must be updated by $\underline{e}_1 \oplus \ldots \oplus \underline{e}_k$ even if the call is not executed. To do so requires a considerably more complex mechanism than we have outlined here. Second, if f writes into global objects, the ℓ and h tables computed by T2 for a program Φ cannot be guaranteed to give lower and upper bounds on the classes of all variables of Φ, whence the execution of Φ cannot be guaranteed secure. Clearly further research is needed to determine whether or not it is feasible to support global variables that are bound dynamically.

4.4.4 Interrupts

Following the approach taken in Section 3.4, we consider the alternative whereby all interrupts are inhibited except those specified by an on statement

$\Phi = \underline{on}\ \zeta\ a\ \underline{do}\ \Phi_1$,

where ζ names an interrupt condition (e.g., overflow or end-file), a is the variable to which it applies, and Φ_1 is the statement to be invoked when the interrupt occurs.

Since execution of Φ_1 is conditioned on (some property of) variable a, the operation push(a) must be performed when Φ_1 is invoked (i.e., when variable a causes an interrupt of type ζ) and pop must be performed when Φ_1 completes and returns. This guarantees that all objects assigned values in Φ_1 are automatically updated by a.

Just as with any other conditional structure, it is necessary to insert update instructions into a program to account for implicit flows caused by the nonoccurrence of an interrupt. As an example, consider the insecure program used earlier, modified to use on statements; it copies a ε {0,1} to b while preserving b = L even when a = H:

```
on overflow x do c := 0;
on overflow y do b := 0;
b := 1;  c := 1;
x := 0;  y := 0;
x := a + MAXVALUE;
y := c + MAXVALUE
```

To eliminate the problem, the program must be modified as follows:

```
on overflow x do c := 0;
on overflow y do b := 0;
b := 1;  c := 1;
x := 0;  y := 0;
x := a + MAXVALUE; push(x); update c; pop;
y := c + MAXVALUE; push(y); update b; pop
```

The example suggests what must be done to deal with the general case. For Φ = "on ζ a do Φ_1", the certification mechanism must process Φ_1 and determine V; it must then identify all points in the program at which "ζ a" can occur and insert there the sequence:

push(a); update V; pop.

(Of course T2 can be used to reduce the update set V, but the principle is the same.) This obviously poses a serious complication for either T1 or T2: these algorithms must now determine if any declared interrupt can occur during the execution of each statement. At present, our research has yielded little insight into this problem. A solution to the dynamic binding case will evidently depend on a viable solution to the interrupt problem.

4.4.5 Static Binding

It is possible to extend the basic hardware and software mechanism to guarantee the secure execution of a program that references some statically bound objects. Let N^s and N^d denote, respectively, the statically and dynamically bound objects. The hardware extensions include an extension of the tag field of a register (or descriptor) for distinguishing objects in N^s from those in N^d. It also includes a modification to the class updating mechanism so that for an object $a \in N^s$, $\underline{a}$ is not changed when information flows into a. It could also verify the validity of the flow; however, this is unnecessary if we assume that some certification program verifies that only valid flows into objects of N^s can occur.

Since algorithm T1 performs no information flow analysis, it is not easily extended to handle the certification of flows into objects of N^s. On the other hand, only two modifications to T2 are needed. (See Figure 4.4-1.) They assume that for an object $a \in N^s$, an initial (fixed) class is specified on the declaration for a, and that this class is assigned to $\ell[a]$ and $h[a]$ (which remain constant). The first modification is to the code for processing assignment statements "$b := f(a_1,\ldots,a_n)$". For $b \in N^s$ it is necessary to verify the security of the explicit flow from the expression to b. This is done by testing that the upper bound on $\underline{a_1} \oplus \ldots \underline{a_n} \oplus \underline{HS}$ flows into $\underline{b}$. The second modification alters the UPDATE procedure responsible for guaranteeing the security of all implicit flows that arise from selection and iteration statements. Here

a) Revised Code for Processing Assignment Statements:

```
$\Phi$ = "$b := f(a_1,\ldots,a_n)$":

  begin
    if $b \in N^d$
      then
        begin
          $\ell[b] := \ell[a_1] \oplus \ldots \oplus \ell[a_n] \oplus SS.\ell$;
          $h[b] := h[a_1] \oplus \ldots \oplus h[a_n] \oplus SS.h$
        end
      else if $h[a_1] \oplus \ldots \oplus h[a_n] \oplus SS.h \not\to \ell[b]$ then
        SECURITY ERROR;
    $V := b$
  end
```

b) Revised Code for UPDATE Procedure:

```
UPDATE: procedure (X, Y: powerset variable);

  $Y := \{b \in X \cap N^d \mid SS.h \not\to \ell[b]\}$;
  for all $b \in Y$ do
    begin $\ell[b] := \ell[b] \oplus SS.\ell$; $h[b] := h[b] \oplus SS.h$ end;
  if $\{b \in X \cap N^s \mid SS.h \not\to \ell[b]\} \neq \emptyset$ then SECURITY ERROR

end UPDATE
```

Figure 4.4-1. Modifications to T2 to Support Static Binding.

it is necessary to verify that the upper bound on HS flows into b for all $b \in N^s$ that are assigned a value in the selection or iteration statement under consideration.

4.4.6 Verify Statement

Consider the following program:

```
max: integer procedure (x, y, z: integer);
  if x ≥ y then max := x else max := y;
  if z > max then max := z
end max.
```

Since the initial values for x, y, and z are not specified, T2 will insert the instruction "update max" after the second if statement so that z → max is guaranteed to hold when the program terminates (x → max and y → max will hold after any execution sequence). Clearly this is not necessary if x = y = z irrespective of the value of x. However, our present mechanism has no provision for specifying such a constraint unless the value of x is known.

Generalizing, we see that the need for update instructions may be reduced, even when the precise initial classes of the input parameters are unknown, if additional information is available about the flow relations that hold on the classes of these variables. We propose that such additional information be declared by means of a new statement

verify $\xi_1,\ldots,\xi_n$,

where each ξ_i ($1 \leq i \leq n$) is a "class expression" specifying constraints on the input classes. Figure 4.4-2 shows the syntax of this statement. The operands of a class expression may be either "absolute" or "relative"; absolute operands reference specific security classes (e.g., L)

```
<class op> ::= ⊕ | ⊗

<flow op> ::=  → | =

<class factor> ::= <security class> | class ( <ident> )

<class term> ::= <class factor> | <class term> <class op> <class factor>

<class exp> ::= <class term> | <class exp> <flow op> <class term>

<verify list> ::= <class exp> | <verify list> , <class exp>

<verify stmt> ::=  verify  <verify list>
```

Figure 4.4-2. Syntax of Verify Statement.

and relative operands reference the class of objects (e.g., *class*(a)). Examples of *verify* statements are:

verify *class*(x) = *class*(y) = *class*(z),

verify *class*(a) = L, *class*(b) → B,

verify *class*(a) ⊕ *class*(b) → *class*(c) = A.

The first of the above examples corresponds to a relation on classes which, if true, obviates the need for the *update* instruction in the earlier program.

The class expressions of a *verify* statement are evaluated on procedure entry, and execution of the procedure is conditioned on all constraints being satisfied. (It might also be possible to use a *verify* statement to control entry to a block of statements, but we have not explored this possibility.)

Although the *verify* statement is appealing, further research is needed to determine its viability. Since precise classes are not known for the input variables, it is necessary to find an alternative method for determining upper and lower bounds on the classes of all variables. One possible approach would be to develop an "algebra" of class expressions. Here the entries in the ℓ and h tables would be class expressions, and T2 would manipulate these entries according to the rules of the algebra. For example, consider the following program:

```
sum: procedure (a,b: integer);
  verify class(a) → class(b);
  sum := a + b
end.
```

To compute ℓ[sum], T2 would form the expression

ℓ[a] ⊕ ℓ[b] ⊕ *SS*.ℓ = *class*(a) ⊕ *class*(b) ⊕ *SS*.ℓ

If SS.ℓ = L initially, this would then be reduced to

class(b)

(since L → class(a) → class(b)). The computation of H[sum] would be the same. We leave for future research the task of determining the feasibility of this approach (or some other).

4.5 Applications

The applications discussed in Section 3.7 for static binding, such as the confinement problem, are also suitable for the mechanism proposed in this chapter. In fact, more flexible solutions to these problems are in principle possible since the security classes of all objects do not have to be bound at compile-time. For instance, the compiler could certify a program for confinement with respect to some of its input parameters, without knowing their precise security classes (which could vary from one call to the next). However, the increased flexibility may be more than offset by the increased cost for the tagged architecture and the more complex certification procedure. Further research is required to resolve these matters.

CONCLUSIONS

Chapter 5

5.1 Summary

In Chapter 2 we developed a model of information flow consisting of three components: an information flow structure specifying security classes and valid flow paths between these classes, states defining possible system configurations, and transition operators specifying state changes. We defined a secure system to be one that did not permit communication along invalid flow paths and showed that security can be obtained by keeping the system in a safe state. An investigation of the mathematical properties of the model led to the conclusion that under certain assumptions, justified by the semantics of the problem, the flow structure forms a lattice. This result is significant in two respects: 1) it permits concise formulations of the security requirements of different systems and 2) it admits efficient implementation mechanisms. It serves as a foundation for the results reported in this thesis.

We examined the security requirements of five increasingly complex systems: 1) single-class, static binding, 2)single-class, dynamic binding, 3) single-class, static and dynamic binding, 4) multi-class, static binding, and 5) multi-class, dynamic binding. The results of this analysis showed that our model provides a unifying theory of secure information flow in that all known systems that restrict information flow fall into these classes. Of particular interest are the two multiclass cases, previously investigated only by Fenton in the context of a Minsky machine [Fe73,Fe74a,Fe74b]. Fenton proposed a hardware mechanism for performing run-time checks during the execution of a program, proved that his mechanism guarantees security under static binding, but conceded that it did not guarantee security under dynamic binding. In Chapters 3 and 4, we made significant extensions to, and improvements on, these results.

In Chapter 3 we introduced a compile-time mechanism for certifying the secure execution of a program under static binding. We formulated the mechanism in terms of semantic actions to be performed during the analysis phase of compilation. The significant contribution of this work is twofold: 1) It permits the security of a program to be established before it executes. This eliminates the need for costly hardware support and/or inefficient software run-time mechanisms. It also eliminates the possibility of programs purposely causing security violations as a means of leaking information. 2) It demonstrates the feasibility and practicality of constructing automatic program certification mechanisms for proving security properties.

In Chapter 4 we investigated the more complex problem of guaranteeing the secure execution of a program under dynamic binding. We outlined a hardware mechanism, based on tagged architecture, for dynamically binding objects to security classes. We presented two possible compile-time mechanisms for transforming an arbitrary program into an equivalent secure one. Both mechanisms operate by inserting instructions into the original program. The first mechanism is considerably less complex than the second, but the second produces better code. The significant contribution of this work is that it proves the feasibility of guaranteeing security under dynamic binding; whether or not it is practical, however, is an open question we leave for future study.

5.2 Areas of Future Research

1. Application. We have not offerred a complete set of guidelines for applying our model and mechanisms to the design of a system. For example, we have not suggested a method for constructing a suitable set of security classes when the usual government and military classification system is not applicable. We have not specified the design of an "operating system kernel" that guarantees the correct implementation of the mechanism. We have said little about the applicability of the mechanism to data base systems.
2. Dynamic Binding. Although we demonstrated the feasibility of certifying the security of a program under dynamic binding, we left many issues unresolved. For example, the question of whether or not it is possible to construct a transformation procedure producing optimal code (in the sense of inserting the minimal number of update instructions) is open. We did not provide an adequate mechanism for dealing with interrupts. We left unresolved the problem of handling global variables. We barely hinted at a method for dealing with the verify statement.
3. Program Development and Debugging. The program certification procedures analyze the flow of information in a program for the purpose of verifying security. The procedures may also be applicable

to a completely different problem: program development and debugging. For example, it would be possible to determine the variables affected by a change to some other variable. It may be possible to determine the best modularization of a program in the sense of minimizing the number of variables common to several modules, while keeping the code expansion due to modularization as small as possible [Ba73].

LIST OF REFERENCES

LIST OF REFERENCES

[Ab69] Abbott, J. C., Sets, Lattices, and Boolean Algebras, Allyn and Bacon, 1969.

[Al70] Allen, F. E., "Control Flow Analysis," Proc. of a Symposium on Compiler Optimization, SIGPLAN Notices, 5, 7, July 1970.

[An74] Andrews, G. R., "COPS - A Protection Mechanism for Computer Systems," Ph.D. Dissertation, Univ. of Wash., July 1974.

[Ba73] Bayer, R., "On Program Volume and Program Modularization," CSD TR 105, Purdue Univ., Sept. 1973.

[BB74] Bell, D. E. and Burke, E. L., "A Software Validation Technique for Certification: The Methodology," MTR-2932, 1, The MITRE Corp., Bedford, Mass., Nov. 1974.

[BL73a] Bell, D. E. and LaPadula, L. J., "Secure Computer Systems: Mathematical Foundations", MTR-2547, ESD-TR-73-278-I, AD-770-768, 1, The MITRE Corp., Bedford, Mass., Mar. 1973.

[BL73b] Bell, D. E. and LaPadula, L. J., "Secure Computer Systems: Mathematical Foundations and Model," M74-244, The MITRE Corp., Bedford, Mass., May 1973.

[Bi67] Birkhoff, G. Lattice Theory, Amer. Math. Soc. Col. Pub., XXV, 3rd ed., 1967.

[CH75] Cash, J., Haseman, W. D., and Whinston, A. B., "Security for the GPLAN System," Purdue Univ., Jan. 1975.

[CD73] Coffman, E. G. and Denning, P. J., Operating System Principles, Prentice Hall, 1973.

[CM72] Conway, R. W., Maxwell, W. L., and Morgan, H. L., "Implementation of Security Structures in Information Systems," Comm. ACM, 15, 4, Apr. 1972, 211-220.

[DD74] Denning, D. E., Denning, P. J., and Graham, G. S., "Selectively Confined Subsystems," Proc. International Workshop on Protection in Operating Systems, IRIA, Aug. 1974.

[De71] Denning, P. J., "Third Generation Computer Systems," Computing Surveys, 3, 4, Dec. 1971.

[De74] Denning, P. J., "Structuring Operating Systems for Reliability," CSD TR 119, Purdue Univ., 1974.

[DV66] Dennis, J. B. and VanHorn, E. C., "Programming Semantics for Multiprogrammed Computations," Comm. ACM, 9, 3, Mar. 1966.

[EL72] Elspas, B., Levit, K., Waldinger, R. and Waksman, A., "An Assessment of Techniques for Proving Program Correctness," Computing Surveys, 4, 2, June 1972, 97-147.

[En72] England, D. M., "Architectural Features of System 250", Infotech State of the Art Report 14, Operating Systems, 1972.

[EL67] Evans, D. C. and LeClerc, J. Y., "Address Mapping and the Control of Access in an Interactive Computer," AFIPS Conf. Proc., 30, SJCC, 1967, 23-30.

[Fa68] Fabry, R. S., "Preliminary Description of A Supervisor for a Machine Oriented Around Capabilities," ICR Quarterly Report 18, Univ. of Chicago, Aug. 1968.

[Fa71] Fabry, R. S., "List Structured Addressing," Ph.D. Thesis, Univ. of Chicago, Mar. 1971.

[Fa73] Fabry, R. S., "Dynamic Verification of Operating System Decisions," Comm. ACM, 16, 11, Nov. 1973, 659-668.

[Fa74] Fabry, R. S., "The Case for Capability Based Computers," Comm. ACM, 17, 6, June 1974.

[Fe73] Fenton, J. S., "Information Protection Systems," Ph.D. Dissertation, Univ. of Cambridge, 1973.

[Fe74a] Fenton, J. S., "Memoryless Subsystems," Computer Journal, 17, 2, Feb. 1974, 143-147.

[Fe74b] Fenton, J. S., "An Abstract Computer Model Demonstrating Directional Information Flow," Univ. of Cambridge, 1974.

[Fu73] Feustel, E. A., "On the Advantages of Tagged Architecture," IEEE Trans. on Computers, C-22, 7, July 1973, 646-656.

[Fl67] Floyd, R. W., "Assigning Meanings to Programs," Math. Aspects of Computer Science, 19, Amer. Math. Soc., (ed. J. T. Schwartz), 1967, 19-32.

[Ga72] Gaines, R. S., "An Operating System Based on the Concept of a Supervisory Computer," Comm. ACM, 15, 3, Mar. 1972, 150-156.

[GS73] George, J. E. and Sagar, G. R., "Variables - Binding and Protection," SIGPLAN Notices, 8, 12, Dec. 1973.

[GD72] Graham, G. S. and Denning, P. J., "Protection - Principles and Practice," AFIPS Conf. Proc., 40, SJCC, 1972, 417-429.

[Gr68] Graham, R. M., "Protection in an Information Processing Utility," Comm. ACM, 11, 5, May 1968, 365-369.

[Gi71] Gries, D., Compiler Construction for Digital Computers, Wiley, 1971.

[Ha70] Hansen, P. B., "The Nucleus of a Multiprogramming System," Comm. ACM, 13, 4, Apr. 1970, 238-250.

[HR75] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D., "On Protection in Operating Systems," Univ. of Ca., Berkeley, Nov. 1974.

[Ho74] Hoare, C. A. R., "Monitors: An Operating System Structuring Concept," Comm. ACM, 17, 10, Oct. 1974.

[Hf69] Hoffman, L. J., "Computers and Privacy: A Survey," Computing Surveys, 1, 2, June 1969, 85-104.

[Hf71] Hoffman, L. J., "The Formulary Model for Flexible Privacy and Access Control," AFIPS Conf. Proc., 39, FJCC, 1971, 587-601.

[IB68] IBM, "System/360 Principles of Operation," IBM Report No. GC28-6821, Sept. 1968.

[IB71] IBM, "System/360 PL/I (F) Language Reference Manual," IBM Report No. GC28-8201-3, 1971.

[Il68] Iliffe, J. K., Basic Machine Principles, MacDonald Elsevier, 1968.

[Jo73] Jones, A. K., "Protection in Programmed Systems," Ph.D. Thesis, Carnegie-Mellon Univ., June 1973.

[La69] Lampson, B. W., "Dynamic Protection Structures," AFIPS Conf. Proc., 35, FJCC, 1969, 27-38.

[La71] Lampson, B. W., "Protection," Proc. Fifth Princeton Symposium on Information Sciences and Systems, Princeton Univ., Mar. 1971, 437-443.

[La73] Lampson, B. W., "A Note on the Confinement Problem," Comm. ACM, 16, 10, Oct. 1973, 613-615.

[Li73] Lindsay, B., "Suggestions for an Extensible Capability-Based Machine Architecture," International Workshop on Computer Architecture, Grenoble, June 1973.

[LM69] Lowry, E. S. and Medlock, C. W., "Object Code Optimization," Comm. ACM, 12, 1, Jan. 1969, 13-22.

[Mi67] Minsky, M. L., Computation; Finite and Infinite Machines, Prentice-Hall, 1967.

[Mo74] Moore, C. G. III, "Potential Capabilities in ALGOL-like Programs," TR 74-211, Dept. of Computer Science, Cornell Univ., Sept. 1974.

[MC74] Moore, C. G. III and Conway, R., "Program Predictability and Data Security," TR 74-212, Dept. of Computer Science, Cornell Univ., Sept. 1974.

[Mr73] Morris, J. H., "Protection in Programming Languages," Comm. ACM, 16, 1, Jan. 1973, 15-21.

[Na63] Naur, P. et.al., "Revised Report of the Algorithmic Language Algol-60," Comm. ACM, 6, 1, 1963.

[Na66] Naur, P., "Proof of Algorithms by General Snapshots," BIT, 6, 4, 1966, 310-316.

[Ne72] Needham, R. M., "Protection Systems and Protection Implementations," AFIPS Conf. Proc., 41, FJCC, 1972, 572-578.

[NF74] Neumann, P. G. and Fabry, R. S., "On the Design of Provably Secure Operating Systems," Proc. of the International Workshop on Protection in Operating Systems, IRIA, Aug. 1974.

[Or72] Organick, E. I., The MULTICS System: An Examination of Its Structure, MIT Press, 1972.

[Po73] Popek, G. J., "Access Control Models," Ph.D. Thesis, Harvard Univ., 1973.

[Po74] Popek, G. J., "Protection Structures," IEEE Computer, June 1974.

[PK74] Popek, G. J. and Kline, C. S., "Verifiable Secure Operating System Software," AFIPS Conf. Proc., 43, NCC, 1974.

[Ro74] Rotenberg, L. J., "Making Computers Keep Secrets," Ph.D. Thesis, MIT, MAC TR-115, Feb. 1974.

[Sc72] Schroeder, M. D., "Cooperation of Mutually Suspicious Subsystems in a Computer Utility," Ph. D. Thesis, MIT, TR-104, Sept. 1972.

[SS72] Schroeder, M. D. and Saltzer, J. H., "A Hardware Architecture for Implementing Protection Rings," Comm. ACM, 15, 3, Mar. 1972.

[Sc71] Scott, D., "The Lattice of Flow Diagrams," Semantics of Algorithmic Languages, Springer Lecture Notes in Math., 188, (ed. E. Engeler), 1971, 311-366.

[St73] Stone, H. S., Discrete Mathematical Structures and Their Applications, Science Research Associates, 1973.

[Su74] Sturgis, H. E., "A Postmortum for a Time Sharing System," Ph.D. Thesis, Univ. of Ca., Berkeley, Jan. 1974.

[Ta74] Tarjan, R., "Finding Dominators in Directed Graphs," SIAM J. Computing, 3, 1, Mar. 1974.

[Ts73] Tsichritzis, D., "Protection Implementation," Proc. of the 7th Annual Princeton Conf. on Information Sciences and Systems, Princeton Univ., Mar. 1973.

[Ts74] Tsichritzis, D., "A Note on Protection in Data Base Systems," Proc. of the International Workshop on Protection in Operating Systems, IRIA, Aug. 1974.

[Wa74] Walter, K. G. et. al., "Modeling the Security Interface," Jennings Computing Center, Case Western Reserve Univ., Report No. 1158, Aug. 1974.

[We69] Weissman, C., "Security Controls in the ADEPT-50 Time-Sharing System," AFIPS Conf. Proc., 35, FJCC, 1969, 119-133.

[Wi72] Wilkes, M. V., Time Sharing Computer Systems, Amer. Elsevier, 2nd ed., 1972.

[Wr71] Wirth, N., "The Programming Language Pascal," Acta Informatica, 1, 1971, 35-63.

[Wu74] Wulf et. al., "HYDRA: The Kernel of a Multiprocessor System," Comm. ACM, 17, 6, June 1974.

VITA

VITA

Dorothy Elizabeth Robling Denning was born in Grand Rapids, Michigan on August 12, 1945. She graduated from Ottawa Hills High School there in 1963. From 1963-1969 she studied mathematics at the University of Michigan, receiving the A.B. degree with distinction in 1967 and the A.M. degree in 1969. While at Michigan, she was employed by the Radio Astronomy Observatory as an Assistant Research Mathematician, by the Computing Center as a Consultant, and by the Computer and Communication Sciences Department as a Teaching Fellow.

From 1969-1972 she was a Systems Programmer at the University of Rochester Computing Center. While at Rochester, she taught a course in compiler design for the Graduate School of Management, and she developed and taught courses in programming languages and compiler design for the Department of Electrical Engineering. She was also an active participant at SHARE meetings, heading the Command Languages Project in 1972.

In 1972 she left Rochester to pursue a Ph.D. degree in Computer Science at Purdue University. From 1972-1974 she was a Graduate Instructor, teaching a course in programming languages. In 1974 she was awarded a Fellowship from IBM. Mrs. Denning has published papers in both Radio Astronomy and Computer Science. She is a member of ACM, Alpha Lambda Delta, and Phi Kappa Phi.