A Lattice Model of Secure Information Flow

Dorothy E. Denning
Purdue University

This paper investigates mechanisms that guarantee secure information flow in a computer system. These mechanisms are examined within a mathematical framework suitable for formulating the requirements of secure information flow among security classes. 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 permit concise formulations of the security requirements of different existing systems and facilitate the construction of mechanisms that enforce security. The model provides a unifying view of all systems that restrict information flow, enables a classification of them according to security objectives, and suggests some new approaches. It also leads to the construction of automatic program certification mechanisms for verifying the secure flow of information through a program.

Key Words and Phrases: protection, security, information flow, security class, lattice, program certification

CR Categories: 4.35

A version of this paper was presented at the Fifth ACM Symposium on Operating Systems Principles, The University of Texas at Austin, November 19–21, 1975.

Work reported herein was supported in part by the National Science Foundation under grants GJ-43176 and GJ-41289 and by IBM under a fellowship. Author's present address: Computer Sciences Department, Purdue University, West Lafayette, IN 47907.

1. Introduction

The security mechanisms of most computer systems make no attempt to guarantee secure information flow. "Secure information flow," or simply "security," means here that no unauthorized flow of information is possible. In the common example of a government or military system, security requires that processes be unable to transfer data from files of higher security classifications to files (or users) of lower ones: not only must a user be prevented from directly reading a file whose security classification exceeds his own, but he must be inhibited from indirectly accessing such information by collaborating in arbitrarily ingenious ways with other users who have authority to access the information [19].

Most access control mechanisms are designed to control immediate access to objects without taking into account information flow paths implied by a given, outstanding collection of access rights. Contemporary access control mechanisms, such as are found in Multics [18, 20] or Hydra [24], have demonstrated their abilities to enforce the isolation of processes essential to the success of a multitask system. These systems rely primarily on assumptions of "trustworthiness" of processes for secure information flow among cooperating processes. Though it is mainly of theoretical interest, Harrison et al. [12] have recently demonstrated that in general it may be undecidable whether an access right to an object will "leak" to a process in a system whose access control mechanism is modeled by an access matrix [11, 15].

In our research into this problem, we sought to find suitable and viable restrictions according to which the security of a system would not only be decidable, but simply so. Our results show that suitable constraints do indeed exist, and moreover within the context of a richly structured model.

2. The Model

2.1 Description

An *information flow model FM* is defined by

$$FM = \langle N, P, SC, \oplus, \rightarrow \rangle.$$

$N = \{a, b, \ldots\}$ is a set of logical storage *objects* or information receptacles. Elements of N may be files, segments, or even program variables, depending on the level of detail under consideration. Each user of the system may also be regarded as an object. $P = \{p, q, \ldots\}$ is a set of *processes*. Processes are the active agents responsible for all information flow.

$SC = \{A, B, \ldots\}$ is a set of *security classes* corresponding to disjoint classes of information. They are intended to encompass, but are not limited to, the familiar concepts of "security classifications," "security categories," and "need to know" [9, 23]. Each object a is bound to a security class, denoted by $\underline{a}$, which specifies the security class associated with the information stored in a. There are two methods of binding objects to security classes: *static binding*, where the security class of an object is constant, and *dynamic binding*, where the security class of an object varies with its contents. Users may be bound, usually statically, to security classes referred to as "security clearances" [2, 22, 23]. Each process p may also be bound to a security class, which we denote by $\underline{p}$. In this case, p may be determined by the security clearance of the user owning p or by the history of security classes to which p has had access.

The *class-combining operator* "$\oplus$" is an associative and commutative binary operator that specifies, for any pair of operand classes, the class in which the result of any binary function on values from the operand classes belongs. The class of the result of any binary function on objects a and b is thus $\underline{a} \oplus \underline{b}$. By extension, the class of the value of an n-ary function $f(a_1, \ldots, a_n)$ is $\underline{a}_1 \oplus \ldots \oplus \underline{a}_n$. To avoid semantic ambiguities that may arise when two different functions over the same domain have overlapping ranges, we assume that the operator "$\oplus$" is independent of the function used to combine values. No generality is lost by this assumption since the effect of a function-dependent "$\oplus$" can be simulated by an appropriate set of processes using a function-independent "$\oplus$" [4]. The set of security classes is closed under "$\oplus$".

A flow relation "$\rightarrow$" is defined on pairs of security classes. For classes A and B, we write $A \rightarrow B$ if and only if information in class A is permitted to flow into class B. Information is said to flow from class A to class B whenever information associated with A affects the value of information associated with B. In this paper we shall be concerned only with flows which result from (sequences of) operations that cause information to be transferred from one object to another (e.g. copying, assignment, I/O, parameter passing, and message sending). This includes flows along "legitimate" and "storage" channels. We shall not be concerned with flows along "covert" channels (i.e. a process's effect on the system load) [16].

The security requirements of the model are simply stated: a flow model FM is *secure* if and only if execution of a sequence of operations cannot give rise to a flow that violates the relation "$\rightarrow$". If a value $f(a_1, \ldots, a_n)$ flows to an object b that is statically bound to a security class $\underline{b}$, then $\underline{a}_1 \oplus \ldots \oplus \underline{a}_n \rightarrow \underline{b}$ must hold. If $f(a_1, \ldots, a_n)$ flows to a dynamically bound object b, then the class of b must be updated (if necessary) so that $\underline{a}_1 \oplus \ldots \oplus \underline{a}_n \rightarrow \underline{b}$ holds for this case also. Assuming that "$\rightarrow$" is transitive, it is easily shown that the security of individual operations implies that of arbitrary sequences of operations [4]. The assumption of transitivity is justified below.

The model we have outlined is a simplified form of the one in [4]. The detailed model accounts for a set of "states" of an underlying computer system and a set of "transition operators" for describing state changes and their associated information flows. It also accounts for such implementation requirements as tags that mark memory locations with the class of information stored in them, or the necessity of nullifying a memory cell when it is deallocated.

2.2 Derivation of Lattice Structure

Under certain assumptions, the model components SC, "$\rightarrow$", and "$\oplus$" form a universally bounded lattice. These assumptions are not arbitrary, but follow from the semantics of information flow. By this we mean either that no generality is lost by them or that they are required for consistency. Consistency means that all flows implied by a permissible flow should also be permitted by the flow relation. For example, if the statement **begin** $b := a$; $c := b$ **end** is secure, then the statement $c := a$ should also be secure. Without a consistent flow structure, it would be possible to masquerade insecure operations as secure ones.

A *universally bounded lattice* is a structure consisting of a finite partially ordered set together with least upper and greatest lower bound operators on the set [3, 21]. To show that $\langle SC, \rightarrow, \oplus \rangle$ forms such a lattice, we establish that:

(1) $\langle SC, \rightarrow \rangle$ is a partially ordered set.
(2) SC is finite.
(3) SC has a lower bound L such that $L \rightarrow A$ for all $A \in SC$.
(4) $\oplus$ is a least upper bound operator on SC.

These assumptions then imply the existence of a greatest lower bound operator on SC, which we denote by "$\otimes$". This in turn implies the existence of a unique upper bound H. Therefore, the structure $\langle SC, \rightarrow, \oplus, \otimes \rangle$ is a lattice with lower bound L and upper bound H.

Assumption (1), that $\langle SC, \rightarrow \rangle$ is a *partially ordered set*, is demonstrated by showing that the relation "$\rightarrow$" is reflexive, transitive, and antisymmetric; that is, for all $A, B, C \in SC$:

(a) $A \rightarrow A$ (reflexive).
(b) $A \rightarrow B$ and $B \rightarrow C \Rightarrow A \rightarrow C$ (transitive).
(c) $A \rightarrow B$ and $B \rightarrow A \Rightarrow A = B$ (antisymmetric).

Reflexivity is required for consistency: since the statement $a := a$ is trivially secure, an inconsistency would exist if $A \not\rightarrow A$ for any $A \in SC$. Transitivity follows from a similar requirement. Since $A \rightarrow B$ implies permission to move a value x from an object in A to one in B, and $B \rightarrow C$ implies it is in turn permissible to move x to an object in C, an inconsistency arises if

$A \not\rightarrow C$. It is easily shown [4] that this assumption implies that a sequence of operations is secure if each operation is individually secure. Antisymmetry follows from the practical assumption of irredundant classes, for $A \rightarrow B$ and $B \rightarrow A$ would imply that anything in one class can be moved into the other, whereupon one of them is unnecessary.

Assumption (2), that the set of security classes SC is finite, is a property of any practical system.

Assumption (3), that there exists a lower bound L on SC, can be made without loss of generality. All constants are candidates for membership in L. However, even if constants must be divided among several classes, L itself need not have any objects assigned to it.

Assumption (4), that the class-combining operator "$\oplus$" is also a *least upper bound* operator, is demonstrated by showing that for all $A,B,C \in SC$:

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

Without property (a) we would have the semantic absurdity that operands could not flow into the class of a result generated from them. Moreover, it would be inconsistent for an operation such as $c := a + b$ to be permitted whereas $c := a$ is not, since the latter operation can be performed by executing the former with $b = 0$. For part (b), consider five objects a, b, c, $c1$, and $c2$ such that $\underline{a} \rightarrow \underline{c}$, $\underline{b} \rightarrow \underline{c}$, and $\underline{c} = \underline{c1} = \underline{c2}$; and consider this program segment:

```
c1 := a;
c2 := b;
c  := c1*c2.
```

Execution of this program segment assigns to c information derived from a and b; therefore, the flow $\underline{a} \oplus \underline{b} \rightarrow \underline{c}$ is implied semantically. For consistency, we require the flow relation to 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.

It is useful to extend the domain of the least upper bound operator to subsets. For a subset X of SC, let $\oplus X$ denote L if X is empty, and the least upper bound of the classes in X otherwise. Then for $n > 1$ and $X = \{A_1, \ldots, A_n\}$, $\oplus X = A_1 \oplus \ldots \oplus A_n$; furthermore, $A_i \rightarrow B (1 \leq i \leq n)$ if and only if $\oplus X \rightarrow B$, or $A_1 \oplus \ldots \oplus A_n \rightarrow B$. This says that information in objects $a_1, \ldots, a_n$ can flow separately into an object b if and only if $\underline{a}_1 \oplus \ldots \oplus \underline{a}_n \rightarrow \underline{b}$ (i.e. the class corresponding to the combination of classes of $a_1, \ldots, a_n$ can flow into the class of b). The least upper bound property also implies that $\oplus$ is associative and commutative; hence $d := (a*b)*c$ is secure if and only if $d := b*(c*a)$ is secure (which is also required for consistency). The highest class H is defined as $\oplus SC$; since it need not have any objects assigned to it, no generality is lost by assuming its existence. Although information can reach H from any other class, it is not allowed ot flow from H to any other class.

Assumptions (1)–(4) imply the existence of a

Fig. 1. Linear ordered lattice.

$SC = \{A_1, \ldots, A_n\}$	A_n
	↑
$A_i \rightarrow A_j$ iff $i \leq j$	A_{n-1}
	↑
$A_i \oplus A_j \equiv A_{\max(i,j)}$	⋮
	↑
$A_i \otimes A_j \equiv A_{\min(i,j)}$	A_2
	↑
$L = A_1$; $H = A_n$	A_1
Description	Representation

greatest lower bound operator on the security classes, which we denote by "$\otimes$". It is shown in [4] that $A \otimes B = \oplus L(A, B)$, where $L(A, B) = \{C | C \rightarrow A$ and $C \rightarrow B\}$. As with the least upper bound "$\oplus$", "$\otimes$" can also be extended to operate on subsets of the security classes SC. For a subset X of SC, let $\otimes X$ be H if X is empty, and otherwise the greatest lower bound of the classes in X. Then for $n > 1$ and $X = \{B_1, \ldots, B_n\}$, $\otimes X = B_1 \otimes \ldots \otimes B_n$; furthermore, $A \rightarrow B_i (1 \leq i \leq n)$ if and only if $A \rightarrow \otimes X$, or $A \rightarrow B_1 \otimes \ldots \otimes B_n$. This says that information in an object a can flow into objects $b_1, \ldots, b_n$ if and only if $\underline{a} \rightarrow \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$. We observe that the lowest class L is just the greatest lower bound of the entire (finite) set of security classes (i.e. $L = \otimes SC$).

2.3 Examples

Figure 1 illustrates that a simple linear ordering on a set of security classes SC satisfies the lattice property. The graphical representation is a standard precedence graph for a partial order, showing only the nonreflexive, immediate relations. This structure is suitable for any system in which the classes are linearly (or hierarchically) ordered. One case is a government or military system in which the security classes are determined solely from the four security levels: *unclassified*, *confidential*, *secret*, and *top secret*. An even simpler case is 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 [5] and also by Fenton (using the names *null* and *priv* for L and H, respectively) [6, 7].

A richer structure satisfying the lattice property is derived from a nonlinear ordering on the set of all subsets of a given finite set X. Figure 2 illustrates this for $X = \{x, y, z\}$. This structure is suitable when X is regarded as a set of properties and classes as combinations of properties from X; information in an object a is not permitted to flow into an object b unless b has at least the properties of a. Consider, for instance, a system that contains medical, financial, and criminal records on individuals (i.e. $X = \{med, fin, crim\}$). Then medical information would be permitted to flow

into only those objects b for which $med \in \underline{b}$, or a combination of medical and financial information would be permitted to flow into only those objects b for which $med \in \underline{b}$ and $fin \in \underline{b}$.

Still richer structures can be constructed as combinations of the two examples above. The profiles of ADEPT [23] form a lattice determined by the (Cartesian) product of two lattices. One lattice is derived from a linear ordering of a set of Authority Levels corresponding to the (unclassified, confidential, etc.) levels of government and military security. The other lattice is derived from an ordering by subsets of the set of all subsets determined by a collection of Categories (properties) corresponding to special control compartments used to restrict access by project and area.

3. Enforcement of Security

The primary difficulty with guaranteeing security lies in detecting (and monitoring) all flow causing operations. This is because all such operations in a program are not explicitly specified—or indeed even executed! 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 operands $a_1, \ldots, a_n$. 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 an expression. To illustrate the difference: the statement **if** $a = 0$ **then** $b := c$ causes an explicit flow from c to b only when $a = 0$ and the assignment to b is performed, but it causes an implicit flow from a to b irrespective of the truth of $a = 0$.

To specify the security requirements of programs causing implicit flows, it is convenient to consider an abstract representation of programs that preserves the flows but not necessarily all of the original structure. An abstract program (or statement) S is defined recursively by:

(1) S is an elementary statement; e.g. assignment or I/O.
(2) There exist S_1 and S_2 such that $S = S_1; S_2$.
(3) There exist $S_1, \ldots, S_m$ and an m-valued variable c such that $S = c{:}S_1, \ldots, S_m$.

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 the value of a variable selects among alternative programs, as abstract programs. Implicit flows can occur only in type (3) structures.

The conditional structure is used to represent all conditional (including iterative) statements found in programming languages. For example, **if** c **then** S_1 **else** S_2 is represented by $c{:}S_1, S_2$. Both **if** c **then** S_1 and **while** c **do** S_1 are represented by $c{:}S_1$, and **do case** c **of** $S_1; \ldots; S_m$ is represented by $c{:}S_1, \ldots, S_m$. When an expression e selects among alternative programs $S_1, \ldots, S_m$, we use the representation $c := e$; $c{:}S_1, \ldots, S_m$. Structures arising from the unrestricted use of **goto** statements can also be represented by the conditional structure, but to do so requires a control flow analysis of the program to determine the set of statements directly conditioned on the values of a variable.

The security requirements for any program of the above form are now stated simply. First, an elementary statement S is secure if any explicit flow caused by S is secure. Specifically, if S replaces the contents of an object b with a value derived from objects $a_1, \ldots, a_n$ ($a_i = b$ for some a_i is possible), then security requires that $\underline{a}_1 \oplus \ldots \oplus \underline{a}_n \rightarrow \underline{b}$ hold after execution of S. If b is dynamically bound to its class, it may be necessary to update $\underline{b}$ when S is executed. Second, $S = S_1; S_2$ is secure if both S_1 and S_2 are individually secure (because of the transitivity of "$\rightarrow$"). Third, $S = c{:}S_1, \ldots, S_m$ is secure if each S_k is secure and all implicit flows from c are secure. Specifically, let $b_1, \ldots, b_n$ be the objects into which S specifies explicit flows (i.e. $i = 1, \ldots, n$ implies that, for each b_i, there is an operation in some S_k that causes an explicit flow to b_i); then all implicit flow is secure if $\underline{c} \rightarrow \underline{b}_i (1 \leq i \leq n)$, or equivalently $\underline{c} \rightarrow \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$, holds after execution of S. If b_i is dynamically bound to its security class, it may be necessary to update $\underline{b}_i$ by $\underline{b}_i := \underline{b}_i \oplus \underline{c}$.

Mechanisms implementing some or all of the security requirements above have been incorporated into ADEPT-50 [23], the MITRE system [2], the Case system [22], Rotenberg's Privacy Restriction Processor [19], Fenton's Data Mark Machine [6, 7, 8], and the security mechanisms proposed by Gat and Saal [10], Jones and Lipton [14], and the author [4]. The following description of these mechanisms distinguishes those supporting only static binding from those supporting both static and dynamic binding.

4. Mechanisms for Static Binding

Mechanisms that enforce security in an environment that supports only static binding of objects to security classes are further characterized by whether they operate at run time or at compile time. The first two mechanisms we shall consider are run-time enforcement mechanisms; the third is a compile-time certification mechanism.

4.1 Access Control Mechanisms

In both the Case system [22] and the MITRE system [2], each process p has an associated clearance class $\underline{p}$ specifying the highest class p can read from (observe) and the lowest class p can write into (modify or extend). Security is enforced by a run-time mechanism that permits p to acquire read access to an object a only if $\underline{a} \rightarrow \underline{p}$, and write access to an object b only if $\underline{p} \rightarrow \underline{b}$. Hence, p can read from $a_1, \ldots, a_m$ and write into $b_1, \ldots, b_n$ only if $\underline{a}_1 \oplus \ldots \oplus \underline{a}_m \rightarrow \underline{p} \rightarrow \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$. This mechanism automatically guarantees the security of all flows, explicit or implicit, since no flow from an object a to an object b can occur unless $\underline{a} \rightarrow \underline{p} \rightarrow \underline{b}$, which implies $\underline{a} \rightarrow \underline{b}$.

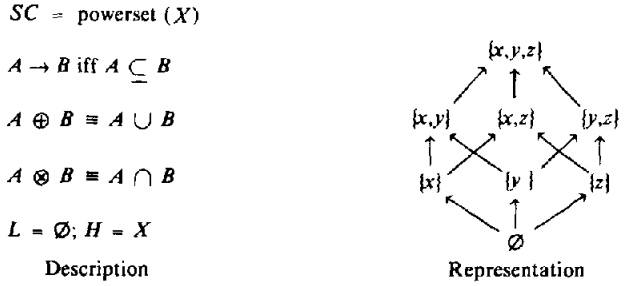
4.2 The Data Mark Machine

Fenton proposed an interesting run-time enforcement mechanism in the context of an abstract computer called a Data Mark Machine [6, 7, 8]. The Data Mark Machine is a Minsky machine [17] extended to include tags (data marks) for binding objects to security classes. Fenton's important observation was that a class $\underline{p}$ should be associated with the program counter of process p. This class is determined as follows: whenever a conditional structure $c{:}S_1, \ldots, S_m$ ($m = 1$ in the Minsky machine) is entered, the current class $\underline{p}$ is pushed onto a stack, and $\underline{p}$ is replaced with $\underline{p} \oplus \underline{c}$; on exit from the structure, $\underline{p}$ is restored by popping the stack. Immediately prior to execution of a statement S conditioned on the values of k condition variables $c_1, \ldots, c_k$, $\underline{p}$ is the least upper bound of the classes of $c_1, \ldots, c_k$ (i.e. $\underline{p} = \underline{c}_1 \oplus \ldots \oplus \underline{c}_k$). If S specifies an explicit flow from objects $a_1, \ldots, a_n$ ($n = 1$ in the Minsky machine) to an object b, the instruction execution mechanism verifies that $\underline{a}_1 \oplus \ldots \oplus \underline{a}_n \oplus \underline{p} \rightarrow \underline{b}$, and inhibits the execution of S if the condition is not satisfied; this automatically checks implicit flows to b (as well as explicit ones) when an explicit flow to b occurs. Fenton proves that this mechanism is also sufficient to insure the security of all implicit flows by proving that under static binding, it is not necessary to verify at run-time implicit flows that occur in the absence of explicit ones. As we shall see later, this result does not hold under dynamic binding.

4.3 Certification Mechanism

In [4] a program certification mechanism is proposed to enforce security. Program certification mechanisms have at least three advantages over run-time enforcement mechanisms. First, the execution of a program is guaranteed to be secure *before* it executes; hence a program cannot leak information by purposely causing security violations. An uncertified program having access to a confidential value x might attempt to convey this value illegally to its owner by causing x security violations; the owner may then be able to obtain the value of x from a record of aborted operations!

Fig. 2. Lattice of subsets of $X = \{x, y, z\}$.



(One solution to this problem is to inhibit all illegal operations without recording their occurrence, or else to record them only after they exceed some limit [6, 19]; this obstructs debugging). The problem is eliminated entirely with a certification mechanism. Second, a certification mechanism 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). Third, 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.

On the other hand, program certification mechanisms have two possible limitations. First, flows not specified by a program cannot be verified. Such a flow could result from a language implementation defect that allows, for example, array bounds to go unchecked or dangling references to occur. Removal of this limitation is possible with adequate language support. Second, a program certified as secure can be transformed by hardware malfunction into an insecure one. Removal of this limitation is possible with a run-time mechanism that doublechecks all flows.

The proposed certification mechanism operates by examining the flow of data through a program to determine its consistency with the flow relation on the given security classes. To make this possible, the programmer must specify (through appropriate declarations) security classes for all objects referenced in the program. The following is an overview of a mechanism detailed in [4].

The lattice properties of the model are exploited to construct an efficient mechanism that is easily incorporated into the analysis phase of a compiler for a high-level language. We shall describe the mechanism in terms of the semantic actions of a compiler that certifies the security of a program having an abstract structure as defined in Section 3.

Consider first an elementary statement S specifying an explicit flow from a value $f(a_1, \ldots, a_n)$ to an object b. To verify that $\underline{a}_1 \oplus \ldots \oplus \underline{a}_n \rightarrow \underline{b}$, the compiler computes the class $\underline{f}(a_1, \ldots, a_n) = \underline{a}_1 \oplus \ldots \oplus \underline{a}_n$ as the objects a_i are recognized; it then verifies that $\underline{f}(a_1, \ldots, a_n) \rightarrow \underline{b}$ when S is recognized. The compiler

also associates with $\underline{S}$ the class $\underline{b}$. For example, if S is the assignment statement $d := e$, where e is the expression $a + b*c$, the compiler would compute $\underline{e} = \underline{a} \oplus \underline{b} \oplus \underline{c}$ during the recognition of e, verify $\underline{e} \rightarrow \underline{d}$ when the assignment is recognized, and set $\underline{S} := \underline{d}$.

Consider next a sequence $S = S_1; S_2$. In this case, the compiler simply sets $\underline{S} := \underline{S}_1 \otimes \underline{S}_2$. Flow relation transitivity implies that no security checks need be performed in this case.

Consider finally a conditional structure $S = c:S_1, \ldots, S_m$. In this case the compiler sets $\underline{S} := \underline{S}_1 \otimes \ldots \otimes \underline{S}_m$; this guarantees that $\underline{S} = \underline{b}_1 \otimes \ldots \otimes \underline{b}_n$, where $b_1, \ldots, b_n$ are the objects information can explicitly flow to in $S_1, \ldots, S_m$. The security of the implicit flows from c to $b_1, \ldots, b_n$ is then checked by verifying $\underline{c} \rightarrow \underline{S}$. For example, if S is the statement **if** c **then begin** $a := 0$; $b := 1$ **end**, the relation $\underline{c} \rightarrow \underline{a} \otimes \underline{b}$ is verified.

5. Mechanisms for Dynamic Binding

A system based purely on dynamic binding is not practical: some objects and most users are usually considered to have a fixed class. Were this not the case in a government or military system, for example, an *unclassified* user would be able to raise his clearance by accessing *top secret* data!

Secure flow into the statically bound objects of a system can be enforced with the mechanisms of Section 4. (Note, however, that a certification mechanism cannot be used if its source objects are dynamically bound). Secure flow into the dynamically bound objects of a system can be enforced with the mechanisms of this section.

There is one intrinsic problem with dynamic updating mechanisms: a change in an object's class may remove that object from the purview of a user whose clearance no longer permits access to the object. The class-change event can thereby be used to leak information, e.g. by removing from the user's purview a file meaning "0".

5.1 Dynamic Data Mark Machine

One form of dynamic binding mechanism is a modified version of Fenton's Data Mark Machine. Updating the class of a dynamically bound object whenever information flows into it follows this simple principle: whenever a statement S specifying a flow from objects $a_1, \ldots, a_n$ to a dynamically bound object b is executed, the class of b is changed, viz., $\underline{b} := \underline{a}_1 \oplus \ldots \oplus \underline{a}_n \oplus \underline{p}$, where $\underline{p}$ is the class of the program counter. (In the case of static binding, the mechanism verifies the relation $\underline{a}_1 \oplus \ldots \oplus \underline{a}_n \oplus \underline{p} \rightarrow \underline{b}$; this mechanism forces the relation to be true by updating $\underline{b}$). That this mechanism alone is insufficient to guarantee security can be seen by considering the execution of the following program (proposed by Fenton [7]).

```
b := c := false;
if ~ a then c := true;
if ~ c then b := true
```

Assuming that the constants *true* and *false* are in the least class L, execution of this program by a process p proceeds as follows:

```
b := c := false;  b := c := L;
p := a;  if ~ a then {c := true;  c := L ⊕ p};  p := L;
p := c;  if ~ c then {b := true;  b := L ⊕ p};  p := L
```

Since $\underline{p} = L \oplus \underline{p}$, this simplifies to:

```
b := c := false;  b := c := L;
if ~ a then {c := true;  c := a};
if ~ c then {b := true;  b := c}
```

When a is *true*, the test "$\sim a$" fails so that $(c, \underline{c})$ remains $(false, L)$; hence the test "$\sim c$" succeeds and $(b, \underline{b})$ becomes $(true, L)$. When a is *false*, the test $\sim a$ succeeds so that $(c, \underline{c})$ becomes $(true, \underline{a})$; hence the test $\sim c$ fails and $(b, \underline{b})$ remains $(false, L)$. Therefore, in both cases the process terminates with $b = a$ yet $\underline{b} = L$. This is because the updating mechanism does not account for the implicit flow that occurs when the statements $c := true$ and $b := true$ are not executed. Therefore, a security violation results unless $\underline{a} = L$.

Fenton [6] and Gat and Saal [10] propose a solution to this problem. In essence, it involves restoring the class and value of any object whose class was increased during execution of a conditional structure to the class and value it had just before entering the structure. Hence, the objects whose class and value are restored behave as "local objects" within the conditional structure. This insures the security of all implicit flows by nullifying those that caused a class increase. For example, in the above program the value of c would be reset to *false* after execution of the statement

if $\sim a$ **then** c : = *true*

if the statement $c := true$ were executed.

In [4] a different solution is proposed. It is based on augmenting the run-time mechanism with a compile-time mechanism. Having performed a data flow analysis of a program to determine what objects could receive an implicit flow without the hardware performing a required class increase, the compiler inserts updating instructions into the compiled program to perform the required class updates. For example, in the program given above, the compiler would insert an instruction at the end of the statement **if** $\sim a$ **then** $c := true$ to update $\underline{c}$ by $\underline{p}$ (i.e. set $\underline{c} := \underline{a}$) before the stack is popped and $\underline{p}$ restored to its value before the **if** statement was executed; hence $\underline{c}$ is guaranteed to be updated to reflect the implicit flow from a even if the assignment to c is not performed.

5.2 Nondecreasing Class Mechanisms

Another type of mechanism is based on the principle of never decreasing the class of an object; that is, if information flows from an object a to an object b, the class of b is always updated by $\underline{b} := \underline{b} \oplus \underline{a}$ (even if the contents of b are replaced by a). Similarly, the class $\underline{p}$ associated with a process is monotonic nondecreasing. This type of mechanism is used by ADEPT [23], Rotenberg's Privacy Restriction Processor [19], and the surveillance and high water mark mechanisms in [14] (the surveillance mechanism permits the class of a storage object to decrease when its contents are replaced by data in a lower class, but not the program counter $\underline{p}$). In Rotenberg's Processor, $\underline{p}$ is determined by the "$\oplus$" of the classes of all objects read (rather than only those of the condition variables involved in implicit flows as in the Data Mark Machine). Hence whenever p writes any information into an object b, its class can be updated by $\underline{b} := \underline{b} \oplus \underline{p}$ to reflect both implicit and explicit flows to b. ADEPT is similar, but $\underline{p}$ is determined by the "$\oplus$" of the classes of all files opened for read or write operations.

ADEPT and Rotenberg's Processor (as described in [19, 23]) suffer the same problem as the dynamic Data Mark Machine: they fail to update the class of an object b when information flows implicitly to b in the absence of any explicit flow. That the class associated with p is nondecreasing has no bearing on this problem. To see why, suppose the example program in Section 5.1 is split between two processes p and q as follows:

```
p: c := false;
   if ~a then c := true
q: b := false;
   if ~c then b := true
```

If p and q are executed sequentially and c is global to both processes, process q terminates with $b = a$, yet $\underline{b} = L$ as before. Even though the classes $\underline{p}$ and $\underline{q}$ of the program counters are nondecreasing, they are independent of each other; $\underline{p}$ is forgotten between the termination of p and the initiation of q.

To solve this problem, an additional mechanism such as described in Section 5.1 is necessary to insure that all implicit flows caused by a process p are secure. For example, using the approach adopted in [4], every object b that could receive an implicit flow without a required class change could have its class updated by $\underline{b} := \underline{b} \oplus \underline{p}$ when p terminates. The mechanisms in [14] resolve this problem by erasing the contents of b if it does not satisfy the relation $\underline{p} \rightarrow \underline{b}$ when p terminates.

Actually ADEPT appears to have an even more fundamental flaw (from our point of view). According to the description in [23], a process p with top secret clearance can write into an existing file with a lower class regardless of the state of $\underline{p}$. Since existing files are statically bound, it can thus transfer data from a top secret file to an unclassified one!

6. Conclusions

The model and mechanisms we have described have many applications. One is confinement: constraining a service process from leaking confidential information about a customer process [16]. The usual solution to this problem is to prevent the service process from retaining any information, confidential or not, after it ceases to operate on behalf of a customer [1, 13, 16]. By controlling the flow of information from confidential to nonconfidential objects, a more flexible solution is possible which permits the service process to save nonconfidential information [4, 5, 7].

Another application is databases. In addition to being able to control the flow of "raw data" in the database to users, it is possible to control the flow of correlations of the data. As an example, consider a database containing a list of names associated with one class and a list of corresponding salaries associated with another. It is possible to separately control information flow about names, salaries, and (name, salary) pairs.

The model does not pretend to address all of the security requirements of a system. Certain requirements that are frequently modeled by an access matrix [11, 15] (e.g. controlling a process's execute access rights to code segments) have been intentionally omitted from the model. On the other hand, by decoupling the right to access information from the right to disseminate it, the flow model goes beyond the access matrix model in its ability to specify secure information flow. A practical system needs both access and flow control to satisfy all security requirements.

Acknowledgments. I wish especially to thank Peter Denning and Butler Lampson; their combined suggestions for improving (and correcting!) the earlier versions of this paper nearly equalled in length that of the paper! I wish also to thank Herbert Schwetman for supervising the thesis research which led to the model and mechanisms described here and Kenneth Omahen for persistently challenging my derivation of the lattice structure. Finally, I wish I knew who the referees were so I could thank them personally!

References

1. Andrews, G.R. COPS—a protection mechanism for computer systems. Ph.D. Th., U. of Washington, July 1974.
2. Bell, D.E., and LaPadula, L.J. Secure computer systems: mathematical foundations and model. M74-244, The MITRE Corp., Bedford, Mass., May 1973.
3. Birkhoff, G. *Lattice Theory*. Amer. Math. Soc. Col. Pub., XXV, 3rd. ed., 1967.
4. Denning, D.E. Secure information flow in computer systems. Ph.D. Th., Purdue U., CSD TR 145, May 1975.
5. Denning, D.E., Denning, P.J., and Graham, G.S. Selectively confined subsystems. Proc. International Workshop on Protection in Operating Systems. IRIA, Aug. 1974, pp. 55–61.
6. Fenton, J.S. Information protection systems. Ph.D. Th., U. of Cambridge, 1973.
7. Fenton, J.S. Memoryless subsystems. *Computer J. 17*, 2 (May 1974), 143–147.
8. Fenton, J.S. An abstract computer model demonstrating directional information flow. U. of Cambridge, 1974.
9. Gaines, R.S. An operating system based on the concept of a supervisory computer. *Comm. ACM 15*, 3 (March 1972), 150–156.
10. Gat, I., and Saal, H.J. Memoryless execution: a programmer's viewpoint. IBM Tech. Rep. 025, IBM Israeli Scientific Center, March 1975.
11. Graham, G.S., and Denning, P.J. Protection—principles and practice. AFIPS Conf. Proc., Vol. 40, 1972 SJCC, AFIPS Press, Montvale, N.J., pp. 417–429.
12. Harrison, M.A., Ruzzo, W.L., and Ullman, J.D. On protection in operating systems. Proc. Fifth Symposium on Operating Systems Principles, The University of Texas at Austin, Nov. 1975, pp. 14–24.
13. Jones, A.K. Protection in programmed systems. Ph.D. Th., Carnegie-Mellon U., June 1973.
14. Jones, A.K., and Lipton, R.J. The enforcement of security policies for computation. Proc. Fifth Symposium on Operating Systems Principles, The University of Texas at Austin, Nov. 1975, pp. 197–206.
15. Lampson, B.W. Protection. Proc. Fifth Princeton Symposium on Information Sciences and Systems, Princeton U., March 1971, pp. 437–443.
16. Lampson, B.W. A note on the confinement problem. *Comm. ACM 16*, 10 (Oct. 1973), 613–615.
17. Minsky, M.L. *Computation; Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, N.J., 1967.
18. Organick, E.I. *The MULTICS System: An Examination of its Structure*, MIT Press, 1972.
19. Rotenberg, L.J. Making computers keep secrets. Ph.D. Th., MIT, MAC TR-115, Feb. 1974.
20. Schroeder, M.D., and Saltzer, J.H. A hardware architecture for implementing protection rings. *Comm. ACM 15*, 3 (March 1972), 157–170.
21. Stone, H.S. *Discrete Mathematical Structures and their Applications*. SRI, Chicago 1973.
22. Walter, K.G., et al. Modeling the security interface. Rep. No. 1158, Jennings Computing Center, Case Western Reserve U., Aug. 1974.
23. Weissman, C. Security controls in the ADEPT-50 time-sharing system. AFIPS Conf. Proc., Vol. 35, 1969 FJCC, AFIPS Press, Montvale, N.J., pp. 417–429.
24. Wulf, W., et al. HYDRA: The kernel of a multi-processor system. *Comm. ACM 17*, 6 (June 1974), 337–345.

Security Kernel Validation in Practice

Jonathan K. Millen
The MITRE Corporation

A security kernel is a software and hardware mechanism that enforces access controls within a computer system. The correctness of a security kernel on a PDP-11/45 is being proved. This paper describes the technique used to carry out the first step of the proof: validating a formal specification of the program with respect to axioms for a secure system.

Key Words and Phrases: validation, verification, correctness, security kernel, formal specification, protection

CR Categories: 4.35, 4.6, 5.24

A version of this paper was presented at the Fifth ACM Symposium on Operating Systems Principles, The University of Texas at Austin, November 19–21, 1975.

This work was supported by the U.S. Air Force under Contract No. F19628-75-C-0001. Author's address: The MITRE Corporation, P.O. Box 208, Bedford, MA 01730.