1976

Certification of Programs for Secure Information Flow

Dorothy E. Denning

Peter J. Denning

Report Number:
76-181

CERTIFICATION OF PROGRAMS FOR SECURE INFORMATION FLOW[1]

Dorothy E. Denning
and
Peter J. Denning[2]

Purdue University

March 1976

CSD-TR 181

Abstract

This paper presents a certification mechanism for verifying the secure flow of information through a program. Because it exploits the properties of a lattice structure among security classes, the procedure is sufficiently simple that it can easily be included in the analysis phase of most existing compilers. Appropriate semantics are presented and proved correct. An important application is the confinement problem: the mechanism can prove that a program cannot cause supposedly non-confidential results to depend on confidential input data.

Key Words and Phrases

protection, security, information flow, program certification, lattice, confinement, security classes

CR Categories

4.35, 4.3, 5.24

---

1 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.

2 Authors' present address: Purdue University, Computer Science Dept., W. Lafayette, Ind. 47907.

1 Introduction

Computer system security relies in part on information flow control, that is, on methods of regulating the dissemination of information among objects throughout the system. An information flow policy specifies a set of security classes for information, a flow relation defining permissible flows among these classes, and a method of binding each storage object to some class. An operation, or series of operations, that uses the value of some object, say x, to derive a value for another, say y, causes a flow from x to y. This flow is admissible in the given flow policy only if the security class of x flows into the security class of y.

Prior work on the enforcement of flow policies has concentrated on run time mechanisms. One type of mechanism enforces a given flow policy by controlling processes' read and write access rights to objects: no process may acquire read access for an input object, or write access for an output object, unless the security class of every input flows into the security class of every output -- even if some outputs depend on only a subset of the inputs. ADEPT-50 [30], the Case system [29], the MITRE system [3, 23], and the Privacy Restriction Processor [26] are of this type. These mechanisms are generally easy to implement because they make no attempt to examine the structure of a program. A second type of (more complex) mechanism accounts for program structures in order to determine flows between specific input and output objects. Fenton's data mark machine [10], the mechanism of Gat and Saal [13], and the surveillance mechanism of Jones and Lipton [19] are of this type. The surveillance mechanism employs a program transformation to insure that all flows are properly accounted for at run time. A detailed discussion of all these mechanisms can be found in [7].

This paper presents a compile time mechanism that certifies a program only if it specifies no flows in violation of the flow policy. Besides the aesthetic attraction of establishing a program's security before it executes, a certification mechanism has important advantages. It can be specified directly in terms of language structures, which facilitates its comprehension and its proof of correctness. It greatly reduces the need for run time checking. It does not impair a program's execution speed. (See also [23]).

Prior certification does not completely eliminate the need for run time checking. Run time support is needed to raise the tolerance against hardware malfunctions and other threats to the integrity of certified programs. It is needed to verify that computed addresses remain in the ranges assumed for them during certification. It is needed to control covert channels, which allow flows outside the storage objects of the system.

2 Lattice Model of Information Flow

We give a brief review of the flow model on which the certification mechanism is based [6, 7]. The model generalizes earlier work as reported in [3, 9, 10, 11, 23, 26, 29, 30].

2.1 Policy Description and Properties

A flow policy can be represented by $\langle S, \rightarrow \rangle$, where S is a given set of security classes and $\rightarrow$ is a flow relation specifying permissible flows between pairs of classes. Each storage object x -- e.g., constant, scalar variable, array, or file -- is assigned (bound) to a security class, denoted by underbar, $\underline{x}$. The notation $\underline{x} \to \underline{y}$ thus means that a flow from object x to object y is permissible in the flow policy. We will suppose that the binding of each object to a security class is static, and can be determined from the declarations contained in a program.

Under the reasonable assumptions that there is a finite number of security classes, that the flow relation is reflexive (i.e., $\underline{x} \to \underline{x}$ is always permissible), and that the flow relation is transitive (i.e., $\underline{x} \to \underline{y} \to \underline{z}$ implies $\underline{x} \to \underline{z}$), we may suppose that $\langle S, \to\rangle$ is a lattice. This means that, corresponding to any pair of classes, there are unique upper and lower bound classes. If $\langle S, \to\rangle$ is not a lattice, it may be transformed into one by adding new classes as necessary without changing the flows among the original classes [8]. The lattice properties are exploited to construct an efficient certification mechanism.

The symbols $\oplus$ and $\otimes$ denote, respectively, the associative and commutative least upper bound and greatest lower bound operators of the lattice $\langle S, \to\rangle$ [4, 28]. The least upper bound is defined so that $\underline{x}_i \to \underline{y}$ for $i = 1,\ldots,m$ is equivalent to the relation $\underline{x}_1 \oplus \ldots \oplus \underline{x}_m \to \underline{y}$. It can be envisaged as requiring that flows from various operand classes must pass through a single, common class en route to a given result class. The greatest lower bound is defined so that $\underline{x} \to \underline{y}_j$ for $j = 1,\ldots,n$ is equivalent to the relation $\underline{x} \to \underline{y}_1 \otimes \ldots \otimes \underline{y}_n$. It can be envisaged as requiring that flows from a given operand class must pass through a single, common class en route to various result classes. There is a highest class H, which is the least upper bound of all classes, and a least class L, which is the greatest lower bound of all classes.

All unnamed programming language constants are members of L. This assumption is reasonable since the flow of an ordinary constant, say "99", into a variable, say x, puts in x no information about any other object. Only when "99" is known to be the value of an object y for which $\underline{y} \nrightarrow \underline{x}$ must its flow be prevented; but this is done by restricting the flow from y, not from "99".

Figures 1 and 2 illustrate lattices that arise frequently in practice. Figure 1 is a linear "priority lattice" on n classes 0,1,...,n-1, where L=0 and H=n-1. This lattice applies to the simple confinement problem with classes nonconfidential (0) and confidential (1) [10] and to the common military security problem with classes unclassified (0), confidential (1), secret (2), and top secret (3) [30]. Figure 2 shows a more complex "property lattice" representing the immediate inclusions among all 2^n subsets of n=3 properties represented as bit vectors. It generalizes easily to any value of n. It is used in systems where information may flow only to a security class having at least the same properties as the originating class [3, 23, 29, 30].

2.2 Flow

Information flows from object x to object y, denoted x => y, whenever information stored in x is transferred to, or used to derive information transferred to, object y. A program statement specifies a flow x => y if execution of the statement could result in a flow x => y.

$S = \{0,1,\ldots,n-1\}$

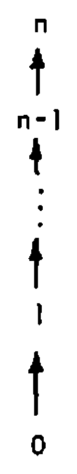
$i \rightarrow j$ iff $i \leq j$

$i \oplus j = \max(i,j)$

$i \otimes j = \min(i,j)$

$L = 0, \quad H = n - 1$

Description



Precedence graph

Figure 1. Linear priority lattice

$S = \{000,001,\ldots,111\}$

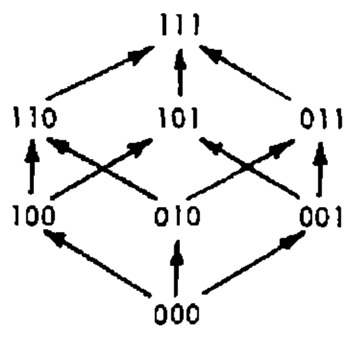
$A \rightarrow B$ iff $OR(A,B) = B$

$A \oplus B = OR(A,B)$

$A \otimes B = AND(A,B)$

$L = 000, \quad H = 111$

Description



Precedence graph

Figure 2. Property lattice for n=3.

Flows are explicit or implicit. An *explicit flow* x => y occurs whenever the operations generating it are independent of the value of x. Assignment statements, I/O statements, and value-returning procedure calls generate explicit flows. An *implicit* flow x => y occurs whenever a statement specifies a flow from some arbitrary z to y, but execution depends on the value of x. Consider for example the statements

y:=1; *if* x=0 *then* y:=0,

where x is either 0 or 1. On termination of these statements, x=y whether or not the *then* clause was executed. Hence the *if* statement causes an implicit flow x => y. In general, all conditional structures generate implicit flows.

It should be noted that the relation => is transitive, that is, x => y => z implies x => z. If x => y because some function having x as an operand stores its result in y, the flow is *direct*; otherwise it is *indirect*. An assignment "y := f (...,x,...)" thus causes flow x => y directly, while the pair "z := f(...,x,...); y := g(...,z,...)" causes flow x => y indirectly.

2.3 Security Requirements

A program p is *secure* if and only if no execution of p results in a flow x => y unless $\underline{x} \rightarrow \underline{y}$. A necessary and sufficient condition for the security of p is then

(1) "x => y for some execution of p only if $\underline{x} \rightarrow \underline{y}$".

Unfortunately, condition (1) is generally undecidable. Any procedure purported to decide it could be applied to the statement

<u>if</u> f(x) halts <u>then</u> y := 0,

and thus provide a solution to the halting problem for an arbitrary recursive function [24]. (In a related study, Harrison, Ruzzo, and Ullman have shown that, without severe restrictions, protection systems contain intractable, if not undecidable, accessing ouestions [16]).

The undecidability is removed if we replace (1) with the security condition

(2) "x => y is specified by p only if $\underline{x} \to \underline{y}$."

The previous <u>if</u> statement can clearly be tested for this condition. However, security condition (2) gives less precision in program certification than (1). For example, consider the program

<u>if</u> x=0 <u>then</u> <u>if</u> x≠0 <u>then</u> y := z

and a flow relation that disallows only z => y. This program is secure by (1) since no execution of it can result in z => y; but it will not be certified by a mechanism based on (2) since it specifies z => y. There is no reason to believe that loss of precision is avoidable; Jones and Lipton, for example, have shown that it is not even possible to construct a mechanism that rejects exactly the insecure executions of a program [19].

The certification mechanism to be presented is based on condition (2). It determines whether a given program specifies invalid flows, irrespective of whether the program can ever execute them.

3 The Certification Mechanism

When the security classes of variables are declared in a program and are static, it is easy to incorporate the certification process into the analysis phase of a compiler. The mechanism will be presented in the form of certification semantics -- actions for the compiler to perform, along with usual semantic actions such as type checking and code generation, when a string of a given syntactic type is recognized. This procedure differs from an information tracing procedure given by Moore [25]: ours verifies program flows against a standard, whereas Moore's seeks primarily to construct a flow graph.

When external objects, such as files and separately compiled procedures, are bound to a program, the linker must verify that the actual security class of each such object corresponds properly to the security class declared formally for it in the program. This must be done before a program is executed.

The certification mechanism exploits lattice properties for efficiency. The transitive flow relation implies that sequences of secure direct flows are secure and, hence, the semantics need only certify the direct flows implied by each syntactic type. The least upper and greatest lower bound properties greatly simplify the amount of information needed to track the origins and destinations of flows. Suppose $x_1,\dots,x_m$ are sources of information for some receiving object y, as in an assignment statement "y := $f(x_1,\dots,x_m)$" or in an output statement "output $x_1,\dots,x_m$ to y". Rather than certify $\underline{x_i} \to \underline{y}$ separately for each i, the compiler may form A = $\underline{x}_1 \oplus \dots \oplus \underline{x}_m$ as the source objects are recognized, and verify simply $A \to \underline{y}$ --

only a single internal variable representing the maximal class of the source objects is needed. Now, suppose $y_1,\dots,y_n$ are to receive information derived from some source object x, as in an input statement "input $y_1,\dots,y_n$ from x", or in a structure generating implicit flows from an object x in a conditional expression to objects y_j in that structure's scope. Rather than certify $\underline{x} \rightarrow \underline{y}_j$ separately for each j, the compiler may form $B = \underline{y}_1 \otimes \dots \otimes \underline{y}_n$ as the receiving objects are being recognized, and verify simply $\underline{x} \rightarrow B$ -- only a single internal variable representing the minimal class of the receiving objects is needed.

The presentation of the full mechanism has been divided into four parts: a) assignment, I/O, and simple control structures; b) general control structures and complex data structures; c) procedure calls; and finally d) exception handling.

3.1 Assignment, I/O, and Simple Control Structures

We consider a programming language that supports only the elementary data types integer, Boolean, and file. Extensions to other types are straightforward. Arithmetic and Boolean expressions are formed from variables and constants as in Pascal [31]. The control structures specify assignment, input and output with files, selection (by an if statement), and iteration (by a while statement). A program comprises a list of declarations, including security class declarations, followed by the executable statements. An example program is given in Figure 3(a).

Table 1 gives the syntax and certification semantics for this language. To avoid ambiguities in the semantics, multiple occurrences of the same syntactic type are distinguished (e.g., <x>, $<x>_1$, and $<x>_2$). The security class of a syntactic type <x> is denoted by $<\underline{x}>$. A compiler variable, CERTIFIED, is initialized to true and set to false if the compiler ever detects a flow specification violating the flow relation. A program is certified as secure if and only if CERTIFIED = true after the entire program has been analyzed. The reader is referred to Gries [15, Sect. 12.2] for an exposition of additional semantic actions, e.g., code generation, that must be defined to complete the compiler.

Figure 4 illustrates the certification of a simple assignment "c := a*2+b". The overall parse can be represented as a syntax tree for the statement. The security classes (in parentheses) are shown opposite each subtree. The semantic actions in effect propagate the security classes of expressions up the tree and verify the flow when the assignment operator is accounted for at the top.

Figure 3(b) shows the certification actions for the example program. When the selection and iteration statements are recognized (lines 20 and 22), the implicit flows from the controlling expressions (the $\oplus$ of the operand classes) to the variables receiving flows in their scopes (the $\otimes$ of all such variable classes) are checked. The example program is certified.

```
 1  begin
 2     i,n: integer security class L;
 3     flag: Boolean security class L;
 4     f1,f2: file security class L;
 5     x,sum: integer security class H;
 6     f3,f4: file security class H;

 7     begin
 8        i := 1;                                  1 → i  (L → L)
 9        n := 0;                                  0 → n  (L → L)
10        sum := 0;                                0 → sum (L → H)
11        while i ≤ 100 do
12           begin
13              input flag from f1;                f1 → flag (L → L)
14              output flag to f2;                 flag → f2 (L → L)
15              input x from f3;                   f3 → x  (H → H)
16              if flag then
17                 begin
18                    n := n + 1;                  n ● 1 → n   (L → L)
19                    sum := sum + x               sum ● x → sum   (H → H)
20                 end;                            flag → n ⊕ sum   (L → L)
21              i := i + 1                         i ● 1 → i   (L → L)
22           end;                                  i ● 100 → flag ⊕ f2 ⊕ x ⊕
                                                     n ⊕ sum ⊕ i   (L → L)
23        output n, sum, sum/n to f4               n ● sum ● sum ⊕ n → f4   (H → H)
24     end
25  end
```

a) Program   b) Certification Checks

Figure 3. A Program and its Certification.

Syntax Rule

Declarations

1 <type> ::= integer | Boolean | file

2 <idlist> ::= <ident> | <idlist> , <ident>

3 <decl> ::= <idlist> : <type> security class

4 <declist> ::= <decl> | <declist> ; <decl>

Expressions

5 <addop> ::= + | - | v

6 <mulop> ::= * | / | ^

7 <relop> ::= < | ≤ | = | ≠ | ≥ | >

8 <var> ::= <ident>

9 <file> ::= <ident>

10 <factor> ::= <var>

11 <factor> ::= <cons>

12 <factor> ::= (<exp>)

13 <factor> ::= - <factor>$_1$

14 <term> ::= <factor>

15 <term> ::= <term>$_1$ <mulop> <factor>

16 <aexp> ::= <term>

17 <aexp> ::= <aexp>$_1$ <addop> <term>

18 <exp> ::= <aexp>

19 <exp> ::= <aexp>$_1$ <relop> <aexp>$_2$

Table I. Basic Certification Semantics.

Certification Semantics

<security class>	for each <ident> in <idlist> associate <security class> with <ident> in the symbo table entry for <ident>

<var> := <ident>

<file> := <ident>

<factor> := <var>

<factor> := L (the least class)

<factor> := <exp>

<factor> := <factor>$_1$

<term> := <factor>

<term> := <term>$_1$ ⊕ <factor>

<aexp> := <term>

<aexp> := <aexp>$_1$ ⊕ <term>

<exp> := <aexp>

<exp> := <aexp>$_1$ ⊕ <aexp>$_2$

Syntax Rule

Assignment

20 <stmt> ::= <var> := <exp>

Input

21 <inlist> ::= <var>

22 <inlist> ::= <inlist>$_1$, <var>

23 <stmt> ::= input <inlist> from <file>

Output

24 <outlist> ::= <exp>

25 <outlist> ::= <outlist>$_1$, <exp>

26 <stmt> ::= output <outlist> to <file>

Compound

27 <stlist> ::= <stmt>

28 <stlist> ::= <stlist>$_1$; <stmt>

29 <stmt> ::= begin <stlist> end

Selection

30 <stmt> ::= if <exp> then <stmt>$_1$ [else <stmt>$_2$]

Iteration

31 <stmt> ::= while <exp> do <stmt>$_1$.

Program

32 <prog> ::= begin <declist> ; <stmt> end

Table 1, cont.

Certification Semantics

$\langle\underline{\text{stmt}}\rangle := \langle\underline{\text{var}}\rangle$
if not ($\langle\underline{\text{exp}}\rangle \rightarrow \langle\underline{\text{var}}\rangle$) then CERTIFIED := $\underline{\text{false}}$

$\langle\underline{\text{inlist}}\rangle := \langle\underline{\text{var}}\rangle$

$\langle\underline{\text{inlist}}\rangle := \langle\underline{\text{inlist}}\rangle_1 \oplus \langle\underline{\text{var}}\rangle$

$\langle\underline{\text{stmt}}\rangle := \langle\underline{\text{inlist}}\rangle$
if not ($\langle\underline{\text{file}}\rangle \rightarrow \langle\underline{\text{inlist}}\rangle$) then CERTIFIED := $\underline{\text{false}}$

$\langle\underline{\text{outlist}}\rangle := \langle\underline{\text{exp}}\rangle$

$\langle\underline{\text{outlist}}\rangle := \langle\underline{\text{outlist}}\rangle_1 \oplus \langle\underline{\text{exp}}\rangle$

$\langle\underline{\text{stmt}}\rangle := \langle\underline{\text{file}}\rangle$

if not ($\langle\underline{\text{outlist}}\rangle \rightarrow \langle\underline{\text{file}}\rangle$) then CERTIFIED := $\underline{\text{false}}$

$\langle\underline{\text{stlist}}\rangle := \langle\underline{\text{stmt}}\rangle$

$\langle\underline{\text{stlist}}\rangle := \langle\underline{\text{stlist}}\rangle_1 \oplus \langle\underline{\text{stmt}}\rangle$

$\langle\underline{\text{stmt}}\rangle := \langle\underline{\text{stlist}}\rangle$

$\langle\underline{\text{stmt}}\rangle := \langle\underline{\text{stmt}}\rangle_1\ [\oplus \langle\underline{\text{stmt}}\rangle_2]$
if not ($\langle\underline{\text{exp}}\rangle \rightarrow \langle\underline{\text{stmt}}\rangle$) then CERTIFIED := $\underline{\text{false}}$

$\langle\underline{\text{stmt}}\rangle := \langle\underline{\text{stmt}}\rangle_1$
if not ($\langle\underline{\text{exp}}\rangle \rightarrow \langle\underline{\text{stmt}}\rangle$ then CERTIFIED := $\underline{\text{false}}$

if CERTIFIED then certify <prog> else report security violation. (CERTIFIED is initialized to $\underline{\text{true}}$ and set to $\underline{\text{false}}$ if a violation is detected)

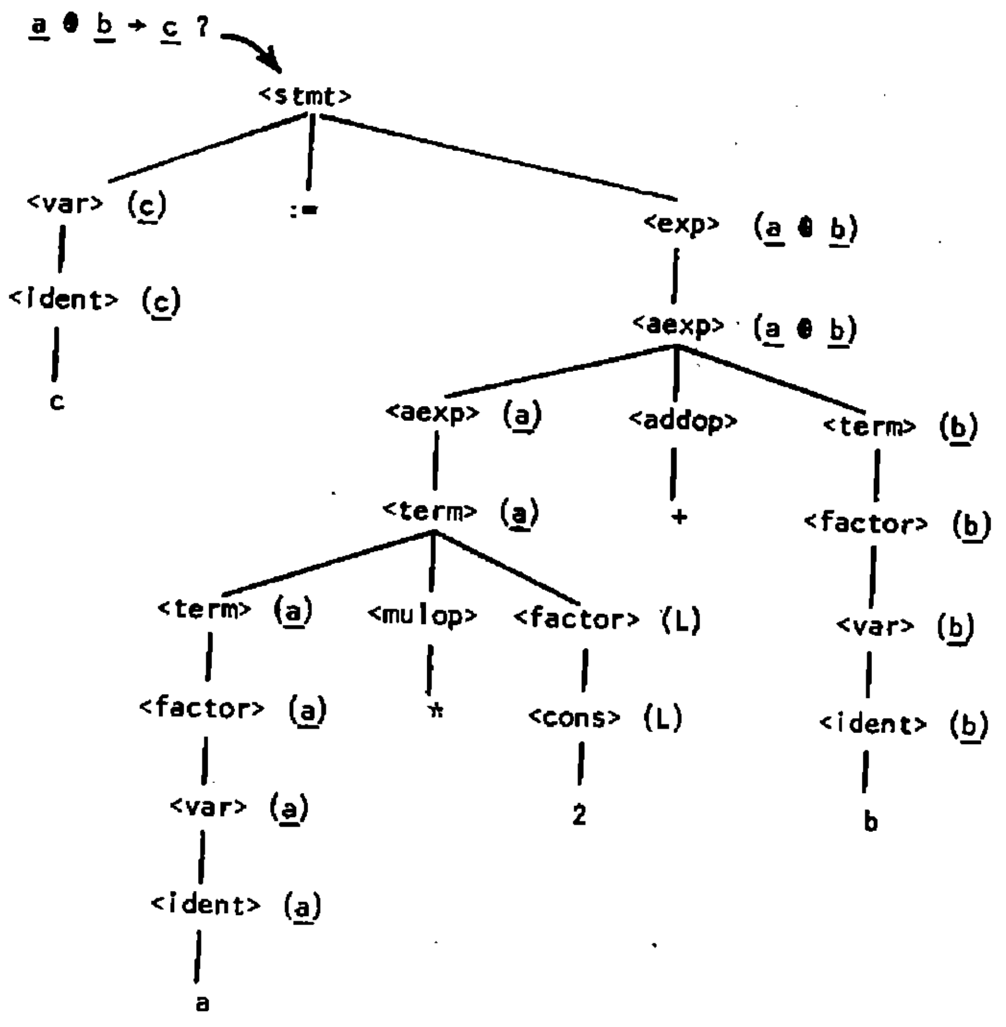


Figure 4. Certification Tree of an Assignment Statement.

The correctness of the certification semantics is straightforwardly established. Let $x_1, \ldots, x_m$ denote the operands (source objects) in an <exp> or an <outlist>, and $y_1, \ldots, y_n$ the results (receiving objects) in an <inlist> or <stmt>. From Table 1, it is easy to deduce that

(p1) $\quad \underline{\text{<exp>}} = \underline{\text{<outlist>}} = \underline{x}_1 \oplus \ldots \oplus \underline{x}_m$

(p2) $\quad \underline{\text{<inlist>}} = \underline{\text{<stmt>}} = \underline{y}_1 \oplus \ldots \oplus \underline{y}_n$

We wish to prove:

Theorem. A program is certified only if it is secure.

The proof is an induction on the structure index i of a given program p; i is simply the number of <stmt> nodes in a syntax tree for p. As a basis, consider i=1. There are three cases for the single simple <stmt> constituting p. 1) Suppose <stmt> = "<var> := <exp>". Let $x_1, \ldots, x_m$ denote the operands of <exp>; by (p1), $\underline{\text{<exp>}} = \underline{x}_1 \oplus \ldots \oplus \underline{x}_m$. The program is certified only if $\underline{\text{<exp>}} \rightarrow \underline{\text{<var>}}$ (Rule 20), and thus only when it is secure. 2) Suppose <stmt> = "input <inlist> from <file>". Let $y_1, \ldots, y_n$ denote the variables in <inlist>; by (p2), $\underline{\text{<inlist>}} = \underline{y}_1 \oplus \ldots \oplus \underline{y}_n$. The program is certified only if $\underline{\text{<file>}} \rightarrow \underline{\text{<inlist>}}$ (Rule 23), and thus only when it is secure. 3) Suppose <stmt> = "output <outlist> to <file>". Let $x_1, \ldots, x_m$ be all the objects in <outlist>; by (p1), $\underline{\text{<outlist>}} = \underline{x}_1 \oplus \ldots \oplus \underline{x}_m$. The program is certified only if $\underline{\text{<outlist>}} \rightarrow \underline{\text{<file>}}$ (Rule 26), and thus only when it is secure. Thus the theorem holds for all programs of one simple statement.

As an induction hypothesis, assume that the theorem holds whenever the program's structure index satisfies $1 \leq i < J$, and consider a program p for which i = J. There are two cases. 1) p is a compound statement of the form <stmt> = "begin <stlist> end." The semantics assume that <stmt> is certified whenever <stlist> is (Rule 29). Since <stlist> denotes a sequence of statements each with index not exceeding J-1, and since the transitivity of the flow relation implies that any sequence of secure flows is secure, <stmt> is secure when <stlist> is. 2) p is a selection or iteration statement of the form <stmt> = "if <exp> then <stmt>_1 [else <stmt>_2]" or "while <exp> do <stmt>_1". Let $x_1,\ldots,x_m$ be the operands of <exp>; by (p1), $\underline{\text{<exp>}} = \underline{x}_1 \oplus\ldots\oplus \underline{x}_m$. Let $y_1,\ldots,y_n$ be the objects receiving flows specified by <stmt>_1 [and <stmt>_2]; by (p2) and Rule 30, $\underline{\text{<stmt>}} = \underline{\text{<stmt>}}_1$ [$\otimes\ \underline{\text{<stmt>}}_2$] = $\underline{y}_1 \otimes \ldots \otimes \underline{y}_n$. By induction <stmt>_1 [and <stmt>_2], having structure indices not exceeding J-1, are certified only if secure. However, Rules 30 and 31 certify <stmt> only if $\underline{x}_1 \oplus \ldots \oplus \underline{x}_m \rightarrow \underline{y}_1 \otimes \ldots \otimes \underline{y}_n$, and thus only when the selection or iteration statement is secure. This completes the correctness proof of the certification semantics.

3.2 General Control and Data Structures

The method of certifying the if and while statements can be extended to any selection or iteration structure expressible as a single statement. This includes, for example, the Pascal repeat, for, and case statements [31]. The principle is to identify the operands $x_1,\ldots,x_m$ of the controlling expression and the objects $y_1,\ldots,y_n$ receiving flows within the scope of the structure, and then verify that $\underline{x}_1 \oplus \ldots \oplus \underline{x}_m \rightarrow \underline{y}_1 \otimes \ldots \otimes \underline{y}_n$.

This technique can be extended to control structures arising from arbitrary goto statements. However, certifying a program with unrestricted gotos requires a control flow analysis of the program to determine the objects receiving flows within the scope of a conditional expression. (This analysis is unnecessary if gotos are restricted -- e.g., to loop exits -- so that the scope of conditional expressions can be determined during syntax analysis). Following is an outline of the analysis required to do the certification. All basic blocks (single-entry, single-exit substructures) are identified. A control flow graph is constructed, showing transitions among basic blocks; associated with block b_i is an expression e_i that selects the successor of b_i in the graph. (How to do this is detailed in [1, 22]). The security class of block b_i is the greatest lower bound of the security classes of all objects receiving flows in b_i (if there are no such objects, this class is H). The immediate forward dominator $IFD(b_i)$ is computed for each block b_i; it is the closest block to b_i among the set of blocks which lie on all paths from b_i to the program exit. Define B_i as the set of all blocks on some path from b_i to $IFD(b_i)$. The security class $\underline{B}_i$ is the greatest lower bound of the classes of blocks in B_i. Since the only blocks directly conditioned on the selector expression e_i of b_i are those in B_i, the program is secure if each block b_i is independently secure and $\underline{e}_i \rightarrow \underline{B}_i$ for all i. Full details of this procedure, with examples, are given in [6].

The mechanism can also be extended to handle complex data structures. We shall consider arrays and records to illustrate the method; Table II shows the semantics. We assume that, just as they are of the same data type, the elements of an array are of the same security class. The certification semantics specify that, as an array reference is processed, the classes of the subscripts should be joined with that of the array, yielding a class

$\underline{\text{<array ref>}} = \underline{\text{<ident>}} \oplus \underline{\text{<sublist>}}$ (Rule 35). This is sufficient as long as the array reference is a source object in an expression. If, however, the array reference is a receiving object, e.g., on the left side of an assignment statement, the relation $\underline{\text{<sublist>}} \rightarrow \underline{\text{<ident>}}$ must also be verified. This is because information about the subscripts flows into the array in this case -- e.g., after the assignment "a[i] := 1" is made on an all-zero array, the value of i can be deduced by searching for the first non-zero element. Since $\underline{\text{<array ref>}} = \underline{\text{<ident>}} \oplus \underline{\text{<sublist>}}$ is computed for any array reference (Rule 35), and since then $\underline{\text{<sublist>}} \rightarrow \underline{\text{<ident>}}$ implies $\underline{\text{<sublist>}} \oplus \underline{\text{<ident>}} = \underline{\text{<ident>}}$, this check reduces to testing whether $\underline{\text{<array ref>}} = \underline{\text{<ident>}}$ when <array ref> is recognized as receiving a flow. We have not shown this check in the certification tables.

As a general rule, certification semantics must generate code that verifies whether computed addresses refer to the objects assumed during certification. Thus the array semantics must verify that the subscripts select elements in the declared range of the array (Rule 35). Without this, a statement like "a[i] :=b" might cause an invalid flow b => c, where c is an object addressed by a[i] when i is out of range.

A record r is a structure comprising fields $x_1,\ldots,x_m$, the i'th element being referenced by the compound name $r.x_i$. Having a distinct name, each element can be assigned to a different security class. The notation $\oplus\underline{r}$ denotes $\underline{r.x_1} \oplus \ldots \oplus \underline{r.x_m}$; $\otimes\underline{r}$ is similarly defined. An operation copying a record from a file f into r is secure only if $\underline{f} \rightarrow \otimes\underline{r}$. An operation copying a record r into a file f is secure only if $\oplus\underline{r} \rightarrow \underline{f}$. An assignment "r := s" for two records of identical structure is secure only if $\underline{s.x_i} \rightarrow \underline{r.x_i}$ for each i. (A stronger, but not equivalent, requirement $\oplus\underline{s} \rightarrow \otimes\underline{r}$ would be easier to implement).

Syntax Rule	
Arrays	
33	<sublist> ::= <exp>
34	<sublist> ::= <sublist>$_1$, <exp>
35	<array ref> ::= <ident> [<sublist>]
Records	
36	<stmt> ::= input <rec> from <file>
37	<stmt> ::= output <rec> to <file>
38	<stmt> ::= <rec>$_1$:= <rec>$_2$

Table II. Certification of Arrays and Records.

Certification Semantics

```
<sublist> := <exp>
<sublist> := <sublist>_1 ⊕ <exp>
<array_ref> := <ident> ⊕ <sublist>
generate subscript range checking code


<stmt> := ⊕ <rec>
if not (<file> → <stmt>) then CERTIFIED := false
<stmt> := <file>
if not (⊕<rec> → <stmt>) then CERTIFIED := false
if <rec>_1 and <rec>_2 have corresponding elements
x_1,...,x_n
   then
      if not (<rec>_1.x_i → <rec>_2.x_i for all i)
         then CERTIFIED := false
      <stmt> := ⊕<rec>_1
   else TYPE ERROR := true
```

3.3 Procedure Calls

A program p is secure only if it calls certified procedures for which the linkage flows are secure. Let q be a procedure with formal input parameters $x_1,\dots,x_m$ and formal output parameters $y_1,\dots,y_n$. Consider a call to q in p of the form

$$\underline{\text{call}}\ q(a_1,\dots,a_m;\ b_1,\dots,b_n),$$

where $a_1,\dots,a_m$ are taken as the actual input parameters and $b_1,\dots,b_n$ as the actual output parameters of the call. The security of the call requires three conditions be verified:

a) q is secure,

b) $\underline{a}_i \rightarrow \underline{x}_i$ for $i = 1,\dots,m$, and

c) $\underline{y}_j \rightarrow \underline{b}_j$ for $j = 1,\dots,n$.

Should the <u>call</u> statement appear in the scope of conditional expressions $e_1,\dots,e_k$, the implicit flows from $e_1,\dots,e_k$, to objects that could receive values during execution of q, must be verified. To this end, the compiler of q must identify all objects $c_1,\dots,c_\ell$ to which q specifies flows; among them will be the formal output parameters of q. The security of the <u>call</u> statement requires that

d) $\underline{e}_1 \oplus \dots \oplus \underline{e}_k \rightarrow \underline{c}_1 \otimes \dots \otimes \underline{c}_\ell$.

If (d) is verified, then by (c) $\underline{e}_1 \oplus \dots \oplus \underline{e}_k \rightarrow \underline{y}_j \rightarrow \underline{b}_j$ for each actual output b_j of q.

Unless p and q are compiled together, conditions (a)-(d) cannot be verified at the same time. However, the certifier can output into the separately compiled p and q information used subsequently by a linker to certify the linkage flows. On recognizing a call to q in p, the certifier outputs the list of m+n+1 classes $(\underline{a}_1,\ldots,\underline{a}_m;\ \underline{b}_1,\ldots,\underline{b}_n;\ \underline{e}_1 \oplus \ldots \oplus \underline{e}_k)$. For procedure q, it outputs the list of m+n+1 security classes $(\underline{x}_1,\ldots,\underline{x}_m;\ \underline{y}_1,\ldots,\underline{y}_n;\ \underline{c}_1 \oplus \ldots \oplus \underline{c}_\ell)$. By matching these lists, the linker can verify conditions (b)-(d).

This mechanism permits constructing a procedure q which outputs results of a higher class than the inputs. This is convenient when q itself, or confidential information used by q to compute its results, must be protected. The flow of information computed by q can be restricted to actual outputs of high security classes.

The foregoing approach poses a serious limitation in designing a procedure q for handling arbitrary classes of information, as is typical of library procedures. The formal inputs $x_1,\ldots,x_m$ must be declared in the highest security class H so that $\underline{a}_i \rightarrow \underline{x}_i$ $(i = 1,\ldots,m)$ can be verified for all calls. This implies that $y_1,\ldots,y_n$ must also be declared in H, since they will be derived from $x_1,\ldots,x_m$. This in turn implies that no call on q can be verified unless the caller has assigned $b_1,\ldots,b_n$ to H, even if $a_1,\ldots,a_m$ are all in the least class L. The foregoing mechanism cannot therefore be used to construct unrestricted procedures that yield low security results from data in arbitrary security classes.

One solution, analogous to the PL/I GENERIC procedure for different data types [17], is to prepare a separate version of q for each possible combination of input security classes. The viability of this approach is questionable when there are many possible combinations of parameter security classes. A more attractive solution results when q is restricted in two ways: its output parameters are derived solely from the input parameters and information in the least class L; it is not permitted to write into any other nonlocal objects. (Local objects can be written if their values are erased when q returns.) The security of a call on such a restricted procedure is verified whenever

a) $\underline{a}_1 \oplus \dots \oplus \underline{a}_m \rightarrow \underline{b}_1 \oplus \dots \oplus \underline{b}_n$, and

b) $\underline{e}_1 \oplus \dots \oplus \underline{e}_k \rightarrow \underline{b}_1 \oplus \dots \oplus \underline{b}_n$.

Table III gives the semantics for certifying these conditions. Note that condition (b) is verified by assigning the class $\underline{b}_1 \oplus \dots \oplus \underline{b}_n$ to the node of the syntax tree associated with the <u>call</u> statement, so that the implicit flow is handled the same as in other statements.

A special case of these restricted procedures is the "function" type procedure (e.g., SQRT, LOG, SIN). Here a procedure f is called during expression evaluation (e.g., by $f(a_1,\dots,a_m)$) and returns with a single result derived entirely from the input parameters and constants. Since there are no explicit output parameters, the function call can be treated as any other expression with operands $a_1,\dots,a_m$. Table III shows the syntax and semantics for this case.

Syntax Rule	
39	<inparams> ::= <exp>
40	<inparams> ::= $<\text{inparams}>_1$, <exp>
41	<outparams> ::= <var>
42	<outparams> ::= $<\text{outparams}>_1$, <var>
43	<stmt> ::= call <ident> (<inparams> ; <outparams>)
44	<fncall> ::= <ident> (<inparams>)
45	<factor> ::= <fncall>

Table III. Certification of Restricted Procedure Cal

Certification Semantics

```
<inparams> := <exp>

<inparams> := <inparams>$_1$ ⊕ <exp>

<outparams> := <var>

<outparams> := <outparams>$_1$ ⊕ <var>

if not <inparams> → <outparams> then
       CERTIFIED := false
<stmt> := <outparams>

<fncall> := <inparams>

<factor> := <fncall>
```

3.4 Exception Handling

Program traps caused by exceptional conditions -- underflow, overflow, divide-by-zero, array subscript range, endfile, and so forth -- require special care [12]. They may cause statements subsequently executed to depend on the variables that caused them. The resulting flows will not be detected by the mechanism defined so far.

The program in Figure 5 will be certified by our mechanism. A problem arises when sum overflows and the trap handler terminates the program: the value of x can be approximated by MAX/LASTi, where MAX is the largest value that can be stored in a register and LASTi is the last value of i entered into file f. The trap has effectively caused a flow of class H information (x) into a class L file (f). Had the programmer indicated the possible loop termination by replacing the while expression e with "not overflow sum", the invalid implicit flow from sum to f would have been detected [5].

One solution -- inhibit all traps -- can be rejected, for it defeats the purpose of traps. Another solution would have the compiler test, for each type of trap possible after each statement, the flow that would arise should that trap occur. This may be rejected for sheer inelegance and impracticality.

A practical solution is based on inhibiting all traps except those for which actions have been defined explicitly by the program. Such definitions could be made with a statement similar to one used in PL/I [17]:

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

```
p:  begin
        i: integer security class L;
        e: Boolean security class L;
        f: file security class L;
        x, sum: integer security class H;
        begin
            sum := 0;
            i := 0;
            e := true;
            while e do
                begin
                    sum := sum + x;
                    i := i + 1;
                    output i to f
                end
        end
    end
```

Figure 5. A program with invalid flow caused by a trap.

where <condition> names a trap condition (underflow, overflow, endfile, etc.), <ident> is the identifier to which the condition applies, and <stmt> contains no gotos. All on statements must appear as part of a program's declaration section. When the trap occurs, <stmt> is executed and control is returned to the point of the trap. Now: suppose there is an on statement "on <condition> y do $<\text{stmt}>_1$", z is a variable receiving a flow in $<\text{stmt}>_1$, another statement $<\text{stmt}>_2$ in the program contains a reference (either read or write) to y, and <exp> is a conditional expression in whose scope $<\text{stmt}>_2$ lies. Since $<\text{stmt}>_1$ is potentially executed immediately after the reference to y in $<\text{stmt}>_2$, the implicit flow $\underline{<\text{exp}>} \rightarrow \underline{z}$ must be verified. To avoid having the compiler backtrack to the on statement to verify $\underline{<\text{exp}>} \rightarrow \underline{z}$, it is simpler to verify a stronger condition: $\underline{y} \rightarrow \underline{z}$ when the on statement is processed, and $\underline{<\text{exp}>} \rightarrow \underline{y}$ when $<\text{stmt}>_2$ is processed. The requires a modification in the semantics: the class of any <stmt> is defined as the greatest lower bound of all $\underline{x}$ such that x either receives a flow, or is an on condition identifier referenced, in <stmt>. Only those traps for which on statements have been declared will be enabled by the compiler.

The program in Figure 5 would be (trivially) certified by this mechanism since it would run with traps inhibited. Had the programmer made clear his intentions via the statement "on overflow sum do e:= false," the program would not be certified.

4. Applications

4.1 The Confinement Problem

A service procedure is confined as long as the system guarantees that it can neither retain any customer information nor encode it into any value transmitted by a storage object [20, 21]. It is selectively confined if this restriction applies only to confidential customer information [5, 10]. Mechanisms enforcing varying degrees of confinement exist or have been proposed [2, 14, 18, 20, 26, 27].

Our certifier is capable of verifying the partial, or total, confinement of a procedure (see Section 3.3). Let p be a procedure with input parameters $x_1,\ldots,x_c,x_{c+1},\ldots,x_m$, and suppose that p is permitted to retain information derived from the nonconfidential inputs $x_1,\ldots,x_c$, but not from the confidential inputs $x_{c+1},\ldots,x_m$. The confinement of p hinges on three properties: 1) p must be internally secure, 2) p must not write into any nonlocal object z for which $\underline{x}_i \rightarrow \underline{z}$ $(c+1 \leq i \leq m)$, and 3) p must invoke only confined procedures. By our definition of security, property (1) implies that confidential information cannot be encoded in supposedly nonconfidential results. Property (2) insures that any information output from p is not derived from confidential inputs. (It does not, however, prevent p from returning confidential results to the customer through the output parameters.) Property (3) requires that p cannot be linked to any other procedure which might violate properties (1) or (2).

4.2 State Variables

Invalid flows ("leaks") can occur in some systems when an observer may examine system state variables and deduce information encoded in them [6, 20, 26]. For example, a process could transmit a confidential value x by locking out files $f_1,\ldots,f_x$; an observer could determine x by counting the number of locked files. These flows can be regulated by associating security classes with all state variables, and verifying flows to and from them as with any other object in the system [21].

4.3 Data Bank Confidentiality

Suppose a system (or network of systems) has a large data base containing different classes of information about individuals. One class might be employment records, another health records, others credit records, tax records, criminal records, and so on. Assuming that all access to the data base must be performed using certified query and update procedures, controlling flows is straightforward. Let each user u have a clearance, i.e., a static security class $\underline{u}$. If u submits a query involving records $x_1,\ldots,x_m$ of the data base, the query procedure would verify $\underline{x}_1 \oplus \ldots \oplus \underline{x}_m \rightarrow \underline{u}$ before accepting the request. Similarly, if u submits an update request for records $y_1,\ldots,y_n$, the update procedure would verify $\underline{u} \rightarrow \underline{y}_1 \otimes \ldots \otimes \underline{y}_n$ before accepting the request.

5 Limitations

Lampson has identified three classes of paths, or "channels", by which processes can transmit information out of their immediate environments [20]. Legitimate channels are the declared, formal outputs of the process; storage channels are other storage objects in the nonlocal environment of the process; and covert channels are any other transmission methods not involving values stored anywhere in the system. Since the first two channels involve information transmitted through storage objects in the system, their flows can be verified by our mechanism. The third, however, employs physical phenomena to connect events within the computer with those outside; examples include program running time, power consumption, noise, and electromagnetic radiation. Flows along these channels are beyond the pale of our certification mechanism. Various run time mechanisms must be used to deal with them. Fenton [9, 10], and Jones and Lipton [19], have shown how to construct mechanisms that prevent an isolated program's running time from depending on confidential information. After a careful analysis, Lipner has concluded that sealing covert channels associated with program running time is at best difficult, and may be impossible in systems of shared resources [21].

Acknowledgements

We are grateful for stimulating discussions with J. S. Fenton, R. S. Gaines, G. S. Graham, S. B. Lipner, J. K. Millen, and H. D. Schwetman. We are particularly grateful to C. Ellison and B. W. Lampson, whose comments on the trap mechanism were most influential.

References

[1] Allen, F. E., "Control Flow Analysis," Proc. of a Symposium on Compiler Optimization, SIGPLAN Notices, 5, 7, July 1970, 1-19.

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

[3] Bell, D. E. and LaPadula, L. J., "Secure Computer Systems: Mathematical Foundations," The MITRE Corp., Bedford, Mass., ESD-TR-73-278, Vol. I-III.

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

[5] Denning, D. E., Denning, P. J., and Graham, G. S., "Selectively Confined Subsystems," Proc. International Workshop on Protection in Operating Systems, IRIA-Laboria (France), August 1974, 55-61.

[6] Denning, D. E., "Secure Information Flow in Computer Systems," Ph.D. Thesis, Purdue Univ., Computer Sci. Dept., May 1975.

[7] Denning, D. E., "A Lattice Model of Secure Information Flow," Comm. ACM 19, 5 (May 1976).

[8] Denning, D. E., "On the derivation of lattice structured information flow policies", Tech Rpt. TR-179, Dept. of Computer Science, Purdue University, (March 1976).

[9] Fenton, J. S., "Information Protection Systems," Ph.D. Dissertation, Univ. of Cambridge, England, Computer Lab, 1973.

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

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

[12] Goodenough, J. B. "Exception handling: Issues and a proposed notation." Comm ACM 18, 12 (Dec. 1975), 683-696.

[13] Gat, I. and Saal, H. J., "Memoryless Execution: A Programmer's Viewpoint," IBM Tech. Rept. 025, IBM Israeli Scientific Center, March 1975.

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

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

[16] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D., "On Protection in Operating Systems," Proc. of the Fifth Symposium on Operating Systems Principles, Nov. 1975, 14-24.

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

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

[19] Jones, A. K. and Lipton, R. J., "The Enforcement of Security Policies for Computation," Proc. of the Fifth Symposium on Operating Systems Principles, Nov. 1975, 197-206.

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

[21] Lipner, S. B., "A Comment on the Confinement Problem," Proc. of the Fifth Symposium on Operating Systems Principles, Nov. 1975, 192-196.

[22] Lowry, E. S. and Medlock, C. W., "Object Code Optimization," Comm. ACM, 12, 1, Jan. 1969, 13-22.

[23] Millen, J. K., "Security Kernel Validation in Practice," Comm. ACM 19, 5 (May 1976).

[24] Minsky, M. L., Computation: Finite and Infinite Machines, Prentice-Hall, 1967.

[25] Moore, C. G. III, "Potential Capabilities in ALGOL-like Programs," TR 74-211, Dept. of Computer Science, Cornell Univ., Sept. 1974.

[26] Rotenberg, L. J., "Making Computers Keep Secrets," Ph.D. Thesis, MIT Project MAC, MAC-TR-115 (Feb. 1974).

[27] Schroeder, M. D., "Cooperation of Mutually Suspicious Subsystems in a Computer Utility," Ph.D. Thesis, MIT Project MAC, MAC-TR-104 (Sept. 1972).

[28] Stone, K. S., Discrete Mathematical Structures and Their Applications, Science Research Associates, 1973.

[29] Walter, K. G. et al.,"Structured Specification of a Security Kernel," Proc. International Conf. on Reliable Software, SIGPLAN Notices, 10, 6 (June 1975), 285-293.

[30] Weissman, C., "Security Controls in the ADEPT-50 Time-Sharing System," Proc. AFIPS 1969 FJCC 35, 119-133.

[31] Wirth, N., "The Programming Language Pascal," Acta Informatica 1, 1 (1971), 35-63.