A Mechanism for Information Control in Parallel Systems

Richard P. Reitman
Syracuse University

Abstract.

Denning and Denning have shown how the information security of sequential programs can be certified by a compile-time mechanism [3]. This paper extends their work by presenting a mechanism for certifying parallel programs. The mechanism is shown to be consistent with an axiomatic description of information transmission.

Keywords and Phrases: information security, information control, concurrency, synchronization, axiomatic logic, consistency.

CR categories: 4.32, 4.35, 5.21, 5.24.

1.0 INTRODUCTION

A secure computer system must protect the information it contains. In particular, the transmission of sensitive information must be controlled. In order to develop practical information security mechanisms, assumptions must be made concerning the type of information transmission that is observable within the system [6]. A common assumption is based on the notion that a programming language defines the set of possible program actions; only those flows of information that can be specified in the programming language are considered [3,8,10]. Accordingly, other flows, such as those that arise from the occurrence of page faults, disk head movement, and program execution time, are considered *covert* [7], and are disregarded.

Since a program's text specifies all flows that can result from program execution, information security can be assured by prohibiting the execution of any program that specifies an undesirable flow of information. A program is said to be *certified* if it has been shown that the program specifies only acceptable flows of information.

This research was supported in part by NSF grant MCS 79-05003.

A deductive *flow logic* that can be used for program certification is presented in [1]. The logic is used to produce flow proofs that capture the flows that arise in both sequential and parallel programs. No practical mechanism based on this theoretical method has been developed to date.

Practical mechanisms have been obtained for systems in which the *security classifications* of objects are static. One of the more general certification mechanisms for static systems has been proposed by Denning and Denning [3]. This mechanism is applicable only to sequential programs that are guaranteed to terminate for all inputs.

The paper is organized as follows. Section two presents the basic concepts necessary to develop information control mechanisms. Section three, which outlines the flow logic, reenforces some of the basic concepts and lays the theoretical foundation needed for section five. Section four discusses the Dennings' mechanism and presents the main result of the paper, the extension of this mechanism to parallel programs. In section five the relationship between flow proofs and the extended mechanism is shown. Section six presents conclusions and areas for future research.

This paper is a synthesis of the work of [3] and [1]. In particular, the mechanism of [3] is extended to parallel programs using the ideas developed in [1]. The extension to parallel programs is an important one, since it is in parallel systems that issues of security are of greatest concern. The correctness of the new mechanism, as well as its relative strength, is demonstrated by proving that a program can be certified using the new mechanism if and only if a flow proof of a restricted form can be developed for the program.

2.0 BASIC CONCEPTS

In this section, the basic concepts needed for the construction of security mechanisms are presented. The discussion of classification schemes closely follows that of [2]. The transmission of information through program execution has been investigated by many researchers [1,3,4,6]. The notion of an information state and of certifying policies in terms of this state is described in [10].

A simple programming language is used throughout the remainder of the paper to specify the set of legal programs. The statements in this language are:

Assignment	x := e
Alternation	<u>if</u> e <u>then</u> S1 <u>else</u> S2
Iteration	<u>while</u> e <u>do</u> S
Composition	<u>begin</u> S1; ...; Sn <u>end</u>
Concurrency	<u>cobegin</u> S1 || ... || Sn <u>coend</u>
Synchronization	wait(sem)
	signal(sem)

By definition, the wait and signal operations on semaphores are indivisable. In addition, each assignment and expression must be executed or evaluated as an indivisable action. As discussed in [9], this requirement may be eliminated if every expression and assignment statement makes at most one reference to a variable that can be changed in another process. In this case, the only requirements are that each wait, signal, and memory reference be an indivisable action.

2.1 Classification Of Information

Program variables contain information. To measure the relative sensitivity of this information a <u>security classification scheme</u> is used. The scheme partitions information into a finite set of equivalence classes and imposes a partial ordering on these classes.

<u>Definition 1</u>.
Given a finite set C and a complete partial order $\leq$ on C, a <u>security classification scheme</u> is the complete lattice (C,$\leq$). High and low are used to denote the maximum and minimum elements of C; $\oplus$ and $\otimes$ are used to denote the least upper bound and greatest lower bound operators respectively.

Every program variable and expression is associated with an <u>information security class</u>. The association between variables and security classes is defined by an <u>information state</u>; this state varies dynamically during program execution.

<u>Definition 2</u>.
Given a security classification scheme (C,$\leq$) and a set of program variables, an <u>information state</u> is a total mapping from the program variables to elements of C. The <u>class</u> of a program variable v, denoted $\underline{v}$, is the element of C associated with v by the information state. The notion of class is extended to expressions by specifying that the class of a constant is low and that the class of e1 <u>op</u> e2 is $\underline{e1} \oplus \underline{e2}$, where <u>op</u> is any arithmetic or Boolean operator.

2.2 Flows Of Information

The assignment of an expression to a variable changes the information that is stored in the variable. The new information contained in the variable has two sources. First, there is a <u>direct flow</u> of information from the expression to the variable. Second, if the assignment is conditionally executed an <u>indirect flow</u> of information occurs.

Indirect flows are either local or global in nature. An indirect flow is said to be local if it is confined to the body of the statment in which the flow is specified. For example, the statement

```
if x = 0 then y := 0 else y := 1
```

transmits information concerning x to the variable y, but not to other variables that are modified elsewhere in the program. This is true because the Boolean condition of the <u>if</u> only affects the execution of the associated <u>then</u> and <u>else</u> parts.

When the effect of an indirect flow is not limited to the body of the statement in which it is generated, the flow is called global. Global flows result from loops and synchronization. For example, the program fragment

```
while x = 0 do y := 0;
z := 1
```

transmits information about x to both y and z. Note that information about x can be inferred by examining z since the assignment z := 1 is executed if and only if x is not equal to zero.

Global flows of information are also produced by parallel programming language constructs. In the simple language of section two, the wait statement for semaphores can produce a global flow. Since the wait statement allows a process to conditionally block, every statement after the wait is executed if and only if a signal was received. For example, the statement

```
cobegin
    if x = 0 then signal(sem)
||
    begin wait(sem); y := 0 end
coend
```

transmits information from x to y, since y is set to zero only if x is zero. Although it is possible for the above program to deadlock (it will if x is not equal to zero), global flows in parallel programs arise not from the possibility of deadlock, but from the synchronization of independent computations. There are programs that cannot deadlock yet transmit information through process synchronization [1].

2.3 Policies And Certification

Information flows between variables as a result of program execution. An <u>information policy</u> is used to indicate which of these flows are acceptable. In particular, the information policy specifies both the set of acceptable information states and the places in the program where this requirement must be satisfied. A program is said to be <u>certified</u> with respect to a policy if and only if it has been shown that every state produced by the program satisfies the policy.

3.0 THE FLOW LOGIC

In [1,10] a deductive logic for reasoning about information flow is introduced. The logic is similar to ones for functional correctness [5,9], except that it deals with *classifications* rather than with *values*. Assertions denote restrictions on the information state; proof rules specify the effect of program execution upon this state. The logical statement {P} S {Q} indicates that if the initial information state satisfies assertion P and statement S terminates, then the final information state satisfies assertion Q. By applying the axioms and rules of the logic, flow proofs of particular logical statements can be produced.

Policy requirements are stated as assertions in the logic. A program is certified by showing that the policy is true at the appropriate places in the flow proof of the program.

3.1 Notation

Assertions in the flow logic may contain the certification variables *local* and *global*; these variables correspond to the two types of indirect information flow. The variable *local* captures indirect flows within a statement; *local* increases when a conditional statement is entered and decreases when it is exited. The variable *global* captures indirect flows between statements that arise from sequencing. Intuitively, it records the information that can be gained by inferring the progress made in executing a composition statement. Thus, *global* increases when a conditionally terminating statement is encountered and never decreases.

The notation {V,L,G} partitions a flow assertion into three parts. V is an assertion about the information state that does not refer to either *local* or *global*. L and G are assertions of the form *local* $\leq$ l and *global* $\leq$ g respectively, where l and g refer to neither *local* nor *global*.

Syntactic substitution and logical derivation are also needed in the flow logic. P[x <- e] denotes the assertion P with every occurrence of the symbol x syntactically replaced by e. P |- Q indicates that using lattice theory and propositional logic Q can be derived from P. Rules of inference in the flow logic are presented as

```
A
---
B
```

where A is the hypothesis and B is the conclusion.

3.2 Proof Rules

This section presents the axioms and proof rules of the flow logic and relates them to the more operational discussion of information flow presented in section two. The axioms and rules are summarized in Figure 1. A more detailed exposition of the logic can be found in [1].

The flow axiom for the assignment statement indicates that the assigned variable receives information from both the expression and certification variables. No new indirect flows are produced.

Alternation statements generate a local flow of information from the Boolean expression to the body of the *then* and *else* parts. This flow is captured in the flow logic by the certification variable *local*. Accordingly, any proof of the alternation statement must reflect that within the *then* and *else* parts *local* has been increased by the class of the booloean e. The requirement V,L,G |- L'[*local* <- *local* $\oplus$ *e*] does this.

There are three aspects of iteration statements that affect information flow. First, since the body S is repeatedly executed, the assertion {V,L',G} must be invariant over the execution of S. Second, the proof of S must take account of the local flow from the Boolean expression e. This is done in a manner analogous to that used for alternation. Third, there is a global flow from e to any statement whose execution is contingent on the termination of the loop. This flow is captured by the requirement V,L,G |- G'[*global* <- *global* $\oplus$ *local* $\oplus$ *e*]. Note that *local* is included since the loop could be nested within an alternation statement.

The rules for composition and consequence are identical to those found in functional correctness. No additional flows need be captured by these rules. Note that the rule for composition captures the interdependence of program statements. In particular, the rule indicates that the flow from one statement is transmitted to the next.

The flow rule for concurrent execution is based on the corresponding rule for functional correctness [9]; a proof of a cobegin statement can be constructed from the proofs of its components only if these proofs are *interference-free*. The definition of non-interfering flow proofs differs slightly from its correctness counterpart since indirect flows in one process do not affect indirect flows in another process.

A semaphore operation is similar to an assignment that increments or decrements a program variable; the semaphore receives additional information due to local and global flows. In addition, a global flow of information due to conditional delay is produced by the wait statement. Accordingly, the axioms for wait and signal are very similar to the axiom for assignment except that the execution of a wait may increase *global*.

Figure 1
The Information Flow Logic

Assignment

$$\{P[\underline{x} \leftarrow \underline{e} \oplus \underline{local} \oplus \underline{global}]\}\ x := e\ \{P\}$$

Alternation

$$\frac{\begin{array}{l}\{V,L',G\}\ S1\ \{V',L',G'\},\ \{V,L',G\}\ S2\ \{V',L',G'\},\\ V,L,G \vdash L'[\underline{local} \leftarrow \underline{local} \oplus \underline{e}]\end{array}}{\{V,L,G\}\ \underline{if}\ e\ \underline{then}\ S1\ \underline{else}\ S2\ \{V',L,G'\}}$$

Iteration

$$\frac{\begin{array}{l}\{V,L',G\}\ S\ \{V,L',G\},\\ V,L,G \vdash L'[\underline{local} \leftarrow \underline{local} \oplus \underline{e}],\\ V,L,G \vdash G'[\underline{global} \leftarrow \underline{global} \oplus \underline{local} \oplus \underline{e}]\end{array}}{\{V,L,G\}\ \underline{while}\ e\ \underline{do}\ S\ \{V,L,G'\}}$$

Composition

$$\frac{\{P0\}\ S1\ \{P1\},\ \ldots,\ \{Pn-1\}\ Sn\ \{Pn\}}{\{P0\}\ \underline{begin}\ S1;\ \ldots;\ Sn\ \underline{end}\ \{Pn\}}$$

Consequence

$$\frac{\{P'\}\ S\ \{Q'\},\ P \vdash P',\ Q \vdash Q'}{\{P\}\ S\ \{Q\}}$$

Concurrent Execution

$$\frac{\{Vi,L,G\}\ Si\ \{Vi',L,G'\},\quad 1 \leq i \leq n \text{ are interference-free}}{\{V1,\ldots,Vn,L,G\}\ \underline{cobegin}\ S1\ \|\ \ldots\ \|\ Sn\ \underline{coend}\ \{V1',\ldots,Vn',L,G'\}}$$

Semaphores

$$\{P[\underline{sem} \leftarrow \underline{sem} \oplus \underline{local} \oplus \underline{global}]\}\ signal(sem)\ \{P\}$$

$$\begin{array}{l}\{P[\underline{sem} \leftarrow \underline{sem} \oplus \underline{local} \oplus \underline{global},\quad wait(sem)\quad \{P\}\\ \quad \underline{global} \leftarrow \underline{sem} \oplus \underline{local} \oplus \underline{global}]\}\end{array}$$

4.0 A MECHANISM FOR STATIC SYSTEMS

This section summarizes the information control mechanism of Denning and Denning [3] and describes an extension to this mechanism. The Concurrent Flow Mechanism (CFM) captures flows that arise from process synchronization and conditional non-termination. An example in which CFM is used to certify a parallel program is also presented.

4.1 The Dennings' Mechanism

Denning and Denning [3] have developed a mechanism for certifying programs when the classification of variables is constant. In this mechanism each variable is statically bound to an acceptable security classification. No program that specifies a violation of this binding is certified.

Definition 3.

Given a security classification $(C,\leq)$ and a program statement S, a static binding sbind for S is a total mapping from the variables, constants, and expressions in S to security classes in C. The static binding of a variable v is denoted by sbind(v), the binding of a constant is low, and the binding of e1 op e2 is sbind(e1) $\oplus$ sbind(e2).*

The mechanism uses simple checks that are performed during compilation to ensure security. Local indirect flows are captured as follows. For each program statement S, mod(S) is the greatest lower bound of all the variables modified in S. If e is an expression whose value controls the execution of S, S is certified only if sbind(e) $\leq$ mod(S). This ensures that flows from e to variables modified by S are permissable.

Global flows are disregarded by the Dennings' mechanism. This shortcoming is caused by the view that the relationships among statements are completely captured by nesting. The proposed extension alleviates this problem by considering the relationships due to statement sequencing.

*The Dennings' original notation has been changed to distinguish between static bindings and current classifications.

4.2 The Concurrent Flow Mechanism (CFM)

The Denning mechanism can be extended to capture flows due to conditional termination and synchronization. First, the classification scheme is extended to include a new smallest element, nil. Next, the function flow(S) is defined. Flow(S) is nil if no global flow is produced by S; otherwise flow(S) is the least upper bound of the global flows produced by S. Finally, the function cert(S) is defined to indicate whether a program S violates a given static binding. These concepts are formally defined as follows:

Definition 4.
Given a classification scheme $(C',\leq')$, the extended classification scheme $(C,\leq)$ associated with $(C',\leq')$ is defined by:
$C = C' \cup \{nil\}$, where nil is not in C', and
$x \leq y$ if and only if either
a. x,y in C' and $x \leq' y$, or
b. x,y in C and x = nil.

Definition 5.
Let $(C,\leq)$ be an extended classification scheme, S be a program statement, and sbind be a static binding for S, then
a. mod(S) is the greatest lower bound of the bindings of variables potentially modified by S,
b. flow(S) is the least upper bound of the global flows produced by S, and
c. cert(S) is true if and only if there is no flow of information specified by S that violates sbind.

S is certified with respect to sbind if and only if cert(S) is true. The appropriate mod, flow, and cert for the simple language of section two are given in Figure 2.

Cert(S) is true only if every component of S is certified. Additional checks are needed for some statements. The checks for assignment and alternation are the same as those originally proposed in [3]. The new check for iteration ensures that no statement in a body of a loop can cause an illegal global flow to a variable modified elsewhere in the loop. This check catches the flow from sem to y in the statement

Figure 2
The Concurrent Flow Mechanism

Statement S	Certification Functions
x := e	mod(S) = sbind(x)
	flow(S) = nil
	cert(S) = sbind(e) $\leq$ sbind(x)
if e	mod(S) = mod(S1) $\otimes$ mod(S2)
then S1	flow(S) = if flow(S1) = flow(S2) = nil
else S2	then nil
	else flow(S1) $\oplus$ flow(S2) $\oplus$ sbind(e)
	cert(S) = cert(S1) and cert(S2) and sbind(e) $\leq$ mod(S)
while e	mod(S) = mod(S1)
do S1	flow(S) = flow(S1) $\oplus$ sbind(e)
	cert(S) = cert(S1) and flow(S) $\leq$ mod(S)
begin	mod(S) = mod(S1) $\otimes \ldots \otimes$ mod(Sn)
S1; ...; Sn	flow(S) = flow(S1) $\oplus \ldots \oplus$ flow(Sn)
end	cert(S) = cert(Si) and cert(Sj) and flow(Sj) $\leq$ mod(Si) $(1 \leq j < i \leq n)$
cobegin	mod(S) = mod(S1) $\otimes \ldots \otimes$ mod(Sn)
S1 || ... || Sn	flow(S) = flow(S1) $\oplus \ldots \oplus$ flow(Sn)
coend	cert(S) = cert(S1) and ... and cert(Sn)
wait(sem)	mod(S) = sbind(sem)
	flow(S) = sbind(sem)
	cert(S) = true
signal(sem)	mod(S) = sbind(sem)
	flow(S) = nil
	cert(S) = true

```
while true do
begin
     y := y + 1;
     wait(sem)
end
```

Note that y is incremented more than once only if the wait statement completes. The new check ensures that sbind(sem) ≤ sbind(y).

The new security check for composition ensures that global flows are acceptable. Using this check the composition of two statements can be certified only if the global flow of the first is no more sensitive than any variable modified by the second. In particular, the statement

```
begin wait(sem); y := 1 end
```

can be certified only if sbind(sem) ≤ sbind(y). Note that parallel composition, unlike sequential composition, does not require an additional certification check since each component statement is executed independently (concurrently executed statements interact through global variables and synchronization primitives).

4.3 An Example

This section examines a program that transmits information through process synchronization. Although the Dennings' mechanism cannot be applied, CFM can be used to certify the program. The example parallel program is presented in Figure 3.

The program transmits information from x to y by ordering process execution. The semaphore modify controls whether m is set to one before or after the assignment y := m. The semaphores modified, read and done ensure that only one process is active at any time. The program has the same effect on x and y as the statement:

```
begin
    m := 0;
    if x = 0
        then begin m := 1; y := m end
        else begin y := m; m := 1 end
end
```

Sequential execution has been enforced for simplicity and to guarantee the flow of information from x to y. If the extra semaphores were eliminated parallel execution could occur but the flow from x to y would depend on the relative execution speed of the processes. Although in this case the flow would not always occur, it could occur and would be considered by CFM.

Note that the program of Figure 3 cannot deadlock and that the final values of the semaphores are the same as their initial values. Therefore, by placing each process in a loop and testing a different bit of x on each iteration an arbitrary amount of information could be transmitted.

Figure 3
Information Flow Using Synchronization

```
var x, y, m : integer;
    modify, modified,
    read, done : semaphore initially(0);

cobegin
    begin
        m := 0;

        if x ≠ 0
            then begin
                signal(modify);
                wait(modified)
            end;

        signal(read);
        wait(done);

        if x = 0
            then begin
                signal(modify);
                wait(modified)
            end;

        wait(done)
    end
||
    begin
        wait(modify);
        m := 1;
        signal(modified)
    end
||
    begin
        wait(read);
        y := m;
        signal(done)
    end
coend
```

Some of the conditions necessary for CFM certification of this program are as follows. First, the if statement is certified only if sbind(x) ≤ sbind(modify). Second, certification of the second process is possible only if sbind(modify) ≤ sbind(m), since the assignment to m occurs after the statement wait(modified). Third, certification of y := m means that sbind(m) ≤ sbind(y). These conditons imply sbind(x) ≤ sbind(y). Note that eliminating the semaphores modified, read, and done would not eliminate this requirement.

5.0 THE RELATIONSHIP OF THE TWO APPROACHES

This section investigates the relationship between the information flow logic and the Concurrent Flow Mechanism. It is shown that a program can be certified using CFM if and only if a flow proof of a restricted form exists. This means that CFM is consistent with the axiomatic description of information transmission. It also indicates that there are programs that can be certified using the flow logic that cannot be certified using CFM.

5.1 Consistency

In order to compare the two approaches, a correspondence between flow assertions and static bindings is needed. A natural correspondence is to associate with a static binding the flow assertion that prohibits the current classification of a variable from exceeding its static binding.

Definition 6.

Given a security classification scheme $(C,\leq)$ and a static binding sbind, the policy assertion corresponding to sbind is the conjunction of all terms of the form $\underline{v} \leq \text{sbind}(v)$, where v is a variable.

CFM is consistent with the flow logic only if for every program certified with respect to a static binding sbind there exists a flow proof that shows that the policy assertion corresponding to sbind is always true. Definition seven precisely states this requirement on flow proofs.

Definition 7.

Given a security classification scheme $(C,\leq)$, a policy assertion I is completely invariant over a program statement S if and only if there exist l,g and g" in C and a flow proof of

$$\{I,\ \underline{\text{local}} \leq l,\ \underline{\text{global}} \leq g\}$$
$$S$$
$$\{I,\ \underline{\text{local}} \leq l,\ \underline{\text{global}} \leq g''\}$$

such that for any statement S' in S, the pre-condition of S' is $\{I, \underline{\text{local}} \leq l', \underline{\text{global}} \leq g'\}$, where l' and g' are elements of C. The above proof is called a completely invariant flow proof for I.

CFM is consistent with the flow logic if a completely invariant policy assertion is necessary for certification. Theorem 1 states that CFM is consistent and indicates how to develop the completely invariant flow proof; the proof of this theorem is given in the Appendix.

Theorem 1.

Suppose $(C,\leq)$ is an extended classification scheme, S is a program statement, sbind is a static binding for the variables in S, and I is the policy assertion corresponding to sbind. S is certified with respect to sbind only if I is completely invariant over S. In particular, for any l and g in C such that $l \oplus g \leq \text{mod}(S)$ there exists a completely invariant flow proof of

$$\{I,\ \underline{\text{local}} \leq l,\ \underline{\text{global}} \leq g\}$$
$$S$$
$$\{I,\ \underline{\text{local}} \leq l,\ \underline{\text{global}} \leq g \oplus l \oplus \text{flow}(S)\}$$

5.2 Relative Strength

There are programs that do not violate their binding but cannot be certified using the Concurrent Flow Mechanism. For example, the program

```
begin x := 0; y := x end
```

cannot be certified with respect to the binding sbind(x) = high, sbind(y) = low by CFM. However, a flow proof of

$$\{\underline{x} \leq \text{high},\ \underline{y} \leq \text{low},\ \text{local} \leq \text{low},\ \text{global} \leq \text{low}\}$$
$$x := 0;$$
$$\{\underline{x} \leq \text{low},\ \underline{y} \leq \text{low},\ \text{local} \leq \text{low},\ \text{global} \leq \text{low}\}$$
$$y := x$$
$$\{\underline{x} \leq \text{low},\ \underline{y} \leq \text{low},\ \text{local} \leq \text{low},\ \text{global} \leq \text{low}\}$$

can be produced. This shows that the policy is never violated by the program. The power of the flow logic is in part its ability to prove intermediate restrictions that are stronger than the policy assertion. In fact, if a flow proof never strengthens the policy assertion CFM certification is possible. Theorem 2 states this precisely; its proof is given in the Appendix.

Theorem 2.

Suppose $(C,\leq)$ is an extended classification scheme, S is a program statement, sbind is a static binding for the variables in S, and I is the policy assertion corresponding to sbind. If I is completely invariant over S, then S can be certified with respect to sbind.

Note that together Theorem 1 and Theorem 2 imply that CFM certification is possible if and only if the policy assertion is completely invariant.

6.0 CONCLUSION

This paper has presented a new mechanism for certifying the information security of programs. This mechanism, CFM, extends the work of Denning and Denning [3] to programming languages that can specify concurrent execution and process synchronization, and also handles conditional termination. The increase in power has been achieved without the loss of computational efficiency; both mechanisms can be computed in time proportional to the length of the program, once the program has been parsed. In addition, it has been shown that the set of programs that can be certified using CFM corresponds exactly to the set of programs for which a restricted form of flow proof exists.

Although CFM is an attractive mechanism when the classifications of objects is fixed, it does not address all information security concerns. Practical mechanisms are needed to ensure information security when object classifications can change dynamically. In addition, program certification is meaningful only if programming language implementations are consistent with the model of information flow used in the mechanisms.

Acknowledgements.

This research has benefited greatly from discussions with Greg Andrews, Carl Hauser, and Nancy McCracken. The comments of Greg Andrews, Anita Jones, and the reviewers helped strengthen the presentation.

Bibliography

1. Andrews, G.R. and Reitman, R.P. An axiomatic approach to information flow in parallel programs. to appear in ACM Trans. on Prog. Languages and Systems.

2. Denning, D.E. A lattice model of secure information flow. Comm. ACM 19,5 (May 1976), 236-243.

3. Denning, D.E. and Denning, P.J. Certification of programs for secure information flow. Comm. ACM 20, 7 (July 1977), 504-513.

4. Fenton, J.S. Memoryless subsystems. Computer Journal 17, 2 (May 1974), 143-147.

5. Hoare, C.A.R. An axiomatic basis for computer programming. Comm. ACM 12, 10 (Oct. 1969), 576-581.

6. Jones, A.K. and Lipton, R.J. The enforcement of security policies for computation. Proc. Fifth Symp. on Operating System Principles (Nov. 1975), 197-206.

7. Lampson, B.W. A note on the confinement problem. Comm. ACM 16, 10 (Oct. 1973), 613-615.

8. London, T.B. The semantics of information flow. Ph.D. Thesis, Cornell University, January 1977.

9. Owicki, S. and Gries, D. An axiomatic proof technique for parallel programs I. Acta Informatica 6 (1976), 319-340.

10. Reitman, R.P. and Andrews, G.R. Certifying information flow properties of programs: an axiomatic approach. Proc. Sixth Symp. on Principles of Programming Languages (Jan. 1979), 283-290.

Appendix

Proofs of Theorem 1 and Theorem 2

Theorem 1.

Suppose (C,$\leq$) is an extended classification scheme, S is a program statement, sbind is a static binding for the variables in S, and I is the policy assertion corresponding to sbind. S is certified with respect to sbind only if I is completely invariant over S. In particular, for any l and g in C such that l $\oplus$ g $\leq$ mod(S) there exists a completely invariant flow proof of

{I, local $\leq$ l, global $\leq$ g}
S
{I, local $\leq$ l, global $\leq$ g $\oplus$ l $\oplus$ flow(S)}

Proof.
It is assumed that cert(S) is true and that l and g are class constants such that l $\oplus$ g $\leq$ mod(S). The proof of the theorem is by induction on the size of the parse tree for S.

Basis.
Let P denote I, local $\leq$ l, global $\leq$ g.

S = x := e. The assumption implies that sbind(e) $\oplus$ l $\oplus$ g $\leq$ sbind(x). By definition P |- e $\leq$ sbind(e), so that P |- e $\oplus$ l $\oplus$ g $\leq$ sbind(x). Since the only term in P that refers to x is x $\leq$ sbind(x), the assignment flow axiom can be applied to construct the desired proof.

S = signal(sem). The proof follows the same line of reasoning used for assignment.

S = wait(sem). P |- sem $\oplus$ l $\oplus$ g $\leq$ sbind(sem) can be shown using an argument similar to that employed for the assignment statement. Since flow(S) is sbind(sem), the axiom for the wait statement can be used to construct the desired flow proof.

Inductive Argument.
Suppose the theorem is true for any statement smaller than S.

S = if e then S1 else S2. The assumption implies that l $\oplus$ g $\oplus$ sbind(e) $\leq$ mod(S). Since S1 and S2 are structurally smaller than S and mod(S) is mod(S1) $\otimes$ mod(S2), proofs of

{I, local $\leq$ l $\oplus$ sbind(e), global $\leq$ g}
Si (i = 1 or 2)
{I, local $\leq$ l, global $\leq$ g$\oplus$l$\oplus$sbind(e)$\oplus$flow(Si)}

exist. By applying the rule for alternation a proof of

{I, local $\leq$ l, global $\leq$ g}
S
{I, local $\leq$ l,
global $\leq$ g$\oplus$l$\oplus$sbind(e)$\oplus$flow(S1)$\oplus$flow(S2)}

can be produced. When flow(S) is not nil the above proof suffices. Otherwise a flow proof showing that global $\leq$ g is preserved by S exists, since flow(S) is nil only if no global flows are produced by S (the proof of this is left to the reader).

S = <u>while</u> e <u>do</u> S1. The assumption implies that flow(S1) $\oplus$ sbind(e) $\leq$ mod(S). Since S1 is structurally smaller than S, a proof showing that

$$\{I,\ \underline{local} \leq l \oplus \text{sbind}(e),\ \underline{global} \leq g \oplus l \oplus \text{sbind}(e) \oplus \text{flow}(S1)\}$$

is invariant over S1 exists. By applying the while rule the desired proof of S can be obtained.

S = <u>begin</u> S1; ... ;Sn <u>end</u>. The assumption implies that flow(Sj) $\leq$ mod(Si) ($1 \leq j < i \leq n$). Since each Si is smaller than S, proofs of

$$\begin{array}{c} \{I, \underline{local} \leq l, \underline{global} \leq g \oplus l \oplus \text{flow}(S1) \oplus \ldots \oplus \text{flow}(Si-1)\} \\ Si \quad (1 \leq i \leq n) \\ \{I, \underline{local} \leq l, \underline{global} \leq g \oplus l \oplus \text{flow}(S1) \oplus \ldots \oplus \text{flow}(Si)\} \end{array}$$

exist. By applying the rule for composition the desired proof can be constructed.

S = <u>cobegin</u> S1 ¦¦ ... ¦¦ Sn <u>coend</u>. The assumption and the inductive hypothesis imply that proofs of

$$\begin{array}{c} \{I,\ \underline{local} \leq l,\ \text{global} \leq g\} \\ Si \quad (1 \leq i \leq n) \\ \{I,\ \underline{local} \leq l,\ \underline{global} \leq g \oplus l \oplus \text{flow}(Si)\} \end{array}$$

exist. Since these proofs are completely invariant, they are also interference-free. Recall that flow(Si) $\leq$ flow(S). Thus by using the flow rules for implication and parallel execution the desired proof for S can be produced.

The following lemma is used in the proof of Theorem 2.

<u>Lemma</u>.
Let (C,$\leq$) be an extended classification scheme, S be a program statement and sbind be a static binding for S. Suppose a proof of

$$\begin{array}{c} \{V,\ \underline{local} \leq l,\ \underline{global} \leq g\} \\ S \\ \{V',\ \underline{local} \leq l,\ \underline{global} \leq g'\} \end{array}$$

exists. If V, <u>local</u> $\leq$ l, <u>global</u> $\leq$ g is satisfied, then

a. $l \oplus g \leq$ mod(S), and
b. $g \oplus$ flow(S) $\leq g'$.

<u>Proof</u>. The proof is by induction on the size of the parse tree for S and by case analysis. The details are left to the reader.

<u>Theorem 2</u>.
Suppose (C,$\leq$) is an extended classification scheme, S is a program statement, sbind is a static binding for the variables in S, and I is the policy assertion corresponding to sbind. If I is completely invariant over S, then S can be certified with respect to sbind.

<u>Proof</u>.
Let P denote the assertion I, <u>local</u> $\leq$ l, <u>global</u> $\leq$ g. It is assumed that a proof of {P} S {I, <u>local</u> $\leq$ l, <u>global</u> $\leq$ g"} exists. The proof of the theorem is by induction on the size of the parse tree for S.

<u>Basis</u>.
S = x := e. Since the assignment axiom can be successfully applied and P contains the term <u>x</u> $\leq$ sbind(x), P ¦- <u>e</u> $\leq$ sbind(x). The only constraint on <u>e</u> by P is <u>e</u> $\leq$ sbind(e), so that <u>e</u> = sbind(e) is possible. Therefore sbind(e) $\leq$ sbind(x) and cert(S) is true.

S = signal(sem). Cert(S) is true by definition.

S = wait(sem). Cert(S) is true by definition.

<u>Inductive Argument</u>.
Suppose the theorem is true for any statement smaller than S.

S = <u>if</u> e <u>then</u> S1 <u>else</u> S2. By the inductive hypothesis, cert(S1) and cert(S2) are true. Let {I, <u>local</u> $\leq$ l', <u>global</u> $\leq$ g'} be the pre-condition of S1 and S2 in the proof of S. The rule for alternation can be successfully applied; therefore P ¦- <u>e</u> $\leq$ l'. Since <u>e</u> = sbind(e) is possible, sbind(e) $\leq$ l'. The lemma, when applied to S1 and S2, implies that l' is bounded above by both mod(S1) and mod(S2). Therefore sbind(e) $\leq$ mod(S) and cert(S) is true.

S = <u>while</u> e <u>do</u> S1. By the inductive hypothesis, cert(S1) is true and an assertion of the form {I, <u>local</u> $\leq$ l', <u>global</u> $\leq$ g'} is invariant over the execution of S1. Since the while rule was applied and <u>e</u> = sbind(e) is possible, P ¦- sbind(e) $\leq$ l'. The lemma implies $l' \oplus g' \leq$ mod(S1) and flow(S1) $\leq$ g'. Therefore flow(S1) $\oplus$ sbind(e) $\leq$ mod(S1), so that cert(S) is true.

S = <u>begin</u> S1; ...; Sn <u>end</u>. By the inductive hypothesis, for every i between 1 and n, cert(Si) is true and a proof of

$$\begin{array}{c} \{I,\ \underline{local} \leq l,\ \underline{global} \leq gi\} \\ Si \\ \{I,\ \underline{local} \leq l,\ \underline{global} \leq gi+1\} \end{array}$$

exists. The lemma, when applied to Si-1 and Si, implies that gi-1 $\oplus$ flow(Si-1) $\leq$ gi $\leq$ mod(Si). Therefore, for every j less than i, flow(Sj) $\leq$ mod(Si) and cert(S) is true.

S = <u>cobegin</u> S1 ¦¦ ... ¦¦ Sn <u>coend</u>. By the inductive hypothesis, each component statement is certified. Therefore, cert(S) is true.