# THE ENFORCEMENT OF SECURITY POLICIES FOR COMPUTATION

Anita K. Jones
Carnegie-Mellon University

and

Richard J. Lipton
Yale University

May, 1975

Jones was supported in part by the National Science Foundation under contract DCR 75-07251 and in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) which is monitored by the Air Force Office of Scientific Research; Lipton was supported in part by the National Science Foundation under contract DCR 74-24193 and in part by the Army Research Office under contract DAHC04-72-A-0001.

ABSTRACT

Security policies define how information within a computer system is to be used. Protection mechanisms are built into these systems to enforce security policies. However, in most systems it is quite unclear what policies a particular mechanism can or does enforce. This paper precisely defines security policies and protection mechanisms in order to bridge the gap between them with the concept of soundness: whether a protection mechanism enforces a specific policy for a given program. Different sound protection mechanisms for the same policy and program can then be compared (on the basis of completeness) to determine if one outperforms the others. We also show that the union of mechanisms for the same policy and program can be taken to produce a more complete mechanism. Although a maximal mechanism exists it cannot necessarily be effectively found.

In addition to developing a theoretical framework in which to discuss security we introduce the surveillance protection mechanism and show both that it is sound and that it is more complete than the commonly used high water mark mechanism.

KEYWORDS AND PHRASES: access control, completeness, high water mark, negative inference, observability, protection, protection mechanism, security, security mechanism, sound, surveillance mechanism, violation notice

CR CATEGORIES: 2.11, 4.30, 4.31

# I. INTRODUCTION

Within computer systems we distinguish between different kinds of information based on a variety of reasons (for example, privacy of individuals which the information describes, laws, cost of theft); in addition, we wish to control how each of the different kinds of information is used. This problem of information control is analogous to one in society, where we wish to control who obtains certain information, the time at which they obtain it as well as which of their subsequent actions are influenced by having the information. Such control of information is difficult to implement in society.* The control of information dissemination has proved to be difficult to implement in computer systems as well and is currently the subject of study by many researchers: [Bell], [Jones], [Lampson], [Popek], [Schroeder], [Walter], [Weissman], [Wulf].

The need for precise and complete understanding of the basic questions is mandatory. To illustrate this, compare security enforcement flaws to compiler design flaws. When a compiler error occurs, the users complain and demand correction. On the other hand, when a security error occurs, the violator does not disclose the system flaw that allowed him to perform prohibited actions. Often in the case of information theft, no trace remains to show that one user read information private to another. For these reasons precision and proofs are not a luxury, but a necessity.

While precision and proofs are required, in order to be credible the basic framework must be simple and clear. No one will believe an unstructured system is secure; just as no one will believe that a formal proof is correct if

---

*A children's tale observes that if one person knows a secret, it's a secret, but if two people know a secret, it's soon public knowledge.

it is too long or poorly structured. The proof of security properties of a computer system is especially sensitive to this issue because such properties span the entire system, not merely a single module of the system. We conclude that to be useful the basis of a theory in the security area must be very simple.

This paper presents a framework in which the underlying principles of security can be investigated. We believe it to be both precise and yet simple. The basic elements of our theory are: a precise definition of a security policy of information control, simple enough so that the ramifications of the policy are clear, and a protection mechanism whose purpose is to 'enforce' a given security policy. Further, we relate these two in terms of soundness and completeness. These terms are discussed informally below.

A security policy, defining what information is to be protected, has a non-procedural form. The definition of a protection mechanism, on the other hand, is procedural.

A protection mechanism is sound provided it enforces the given security policy. Thus soundness is the bridge between the non-procedural security policy and the procedural protection mechanism. Currently this relation is unclear in existing security systems. When a software designer is asked to define the security policies his protection mechanism can enforce his answer is phrased in terms of a procedural description of some of the possible state changes of his mechanism. Such a description is insufficient. A security policy must be expressed in a language such that the policy's author is convinced it is what he intended.

While the above discussion suggests that soundness is a binary relation between protection mechanisms and security policies, such is not the case. It also depends on just what attributes of a program's execution are observable,

i.e. visible to outside observers. For example an outside observer may or may not be able to detect the running time of a program's execution. Thus there are protection mechanisms which are sound only provided running time is not observable.

Completeness is a measure of how well a given protection mechanism performs. Sound protection mechanisms abound; the key is to find those that are complete in that they allow the user to do as much as possible. The importance of completeness is unclear in current literature; here it has central importance.

One way to evaluate a framework such as the one developed here is to see what results have come as a consequence of the theory. There are four principal results. First, we have developed a new protection mechanism called the surveillance protection mechanism. Second, we can faithfully represent parts of actual security systems in this framework. Third, we can show that the surveillance mechanism is more complete than several other existing protection mechanisms. Fourth, we can show that there exists a maximal protection mechanism. Though the union of mechanisms can be used to derive increasingly powerful mechanisms, the maximal protection mechanism cannot necessarily be effectively discovered.

## II. BASIC MODEL

The first concept to be defined is the concept of a computer program. We define Q to be a program provided

$$Q: D_1 x \ldots x D_k \rightarrow E_1 x \ldots x E_n$$

where $D_i$ is the range of the $i^{th}$ input and $E_j$ the range of the $j^{th}$ output. $Q(x_1,\ldots,x_k)$ denotes the n-tuple that corresponds to the input $(x_1,\ldots,x_k)$. If $Q(x_1,\ldots,x_k) = y_1,\ldots,y_n$ we use $Q_i(x_1,\ldots,x_k)$ to denote $y_i$ for i from 1 to n.

Programs defined here can be viewed in many different ways. A program could be a 'cosine' subroutine with a single input parameter $x_1$, specifying an angle in degrees, and a real valued output $\cos(x_1)$. Alternatively, Q could be an entire user job (perhaps composed of several processes) with inputs $x_1,\ldots,x_{k-1}$ which are files and input $x_k$ a stream of characters from a terminal. Output could then be the files $y_1,\ldots,y_{n-1}$ and the terminal output $y_n$. An important feature, therefore, of this definition of program is that it is independent of any particular model or grain of computation.

We now turn our attention to the study of protection mechanisms.

Definition. Suppose that $Q: D_1 x \ldots x D_k \rightarrow E_1 x \ldots x E_n$ is a program. Then M is a protection mechanism for Q provided $M: D_1 x \ldots x D_k \rightarrow E_1' x \ldots x E_n'$ and $E_i' = E_i \cup F$ where F is disjoint from $E_i$ for all i, and for all $x_1,\ldots,x_k$ and $1 \leq i \leq n$

$$M_i(x_1,\ldots,x_k) = Q_i(x_1,\ldots,x_k) \text{ or } M_i(x_1,\ldots,x_k) \in F.$$

The set F consists of the violation notices of M.

A protection mechanism acts as follows: For input $\bar{a}$* and for each i the mechanism decides whether or not to give the output value $Q_i(\bar{a})$. If it decides not to allow this output, then the protection mechanism gives a violation notice from F. The key constraint on the protection mechanism M is that it must act simply as a "gatekeeper" (see Figure 1); it can allow the normal output or an output from F. Intuitively violation notices, i.e., elements from F, can be thought of as alleged attempted violations. The definition of protection mechanism is quite general in that the protection mechanism can decide whether or not to output $Q(\bar{a})$, based on any criterion at all. Note that any program satisfying these constraints is a protection mechanism. We make no assumption about whether it executes, interprets or even simulates the program Q.

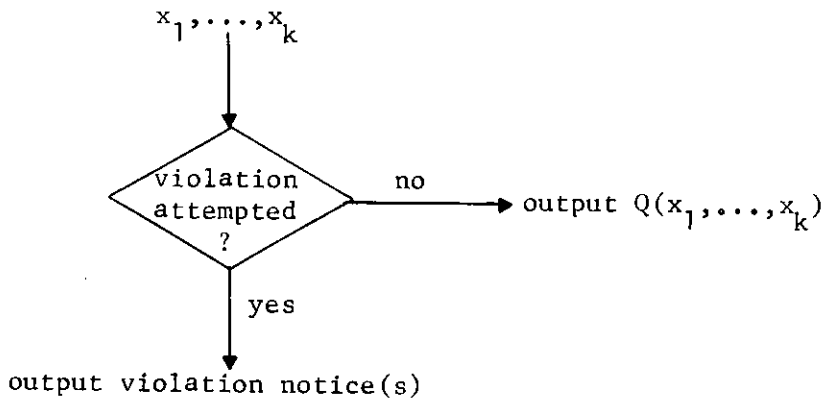



Figure 1. A functional view of a protection mechanism for Q.

More concretely, suppose that a user submits a job to a computer system, and then waits for his output from a line printer in a batch environment or from an interactive terminal in a time sharing environment. In an unprotected

*$\bar{a}$ denotes $a_1,\ldots,a_k$ the actual input values for input variables $x_1,\ldots,x_k$ respectively; the value of k will always be clear from the context.

computer system the user will always receive $Q(\bar{a})$, that is, the result of his computation. In a protected computer system he may or may not receive $Q(\bar{a})$: if his computations violated no 'rules', then he will receive $Q(\bar{a})$; otherwise, he will receive a violation notice.

It is important to note that M always outputs something. This restriction that M be total is an entirely practical one. In any real computer system, programs are never allowed to run forever. Moreover, at the cost of some complexity in our theory we could extend the concept of program to partial functions, i.e. to programs that did not always halt. However, the cost seems to outweigh the advantages; therefore, in this paper all programs will be assumed to be total.

The purpose of protection mechanism is to 'control information flow'; in our framework this means to enforce security policies.

Definition. A security policy $I_j$ for the $j^{th}$ output of program $Q: D_1 x \ldots x D_k \rightarrow E_1 x \ldots x E_n$ is a subset of $x_1, \ldots, x_k$.

We will refer to the set of security policies $I_1, \ldots, I_n$ for the outputs of a program to be the security policy I of the program.

Definition. The protection mechanism M is sound for program $Q: D_1 x \ldots x D_k \rightarrow E_1 x \ldots x E_n$ and security policy I provided for each output $y_j$ and associated policy $I_j = (x_{j_1}, \ldots, x_{j_m})$ there exists a program $M': D_{j_1} x \ldots x D_{j_m} \rightarrow E_1' x \ldots x E_n'$ such that

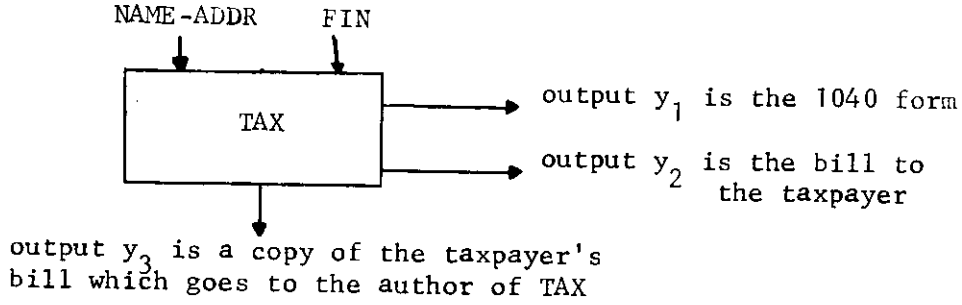
$$\forall x_1 \ldots \forall x_k \quad M_j(x_1, \ldots, x_k) = M_j'(x_{j_1}, \ldots, x_{j_m}).$$

Intuitively the meaning of the security policy $I_j$ for an output variable is: allow the output to depend on inputs from $I_j$ but not on any inputs which

lie outside of $I_j$. Therefore the protection mechanism M is sound for program Q and security policy I provided that for each j, the $j^{th}$ output of M only depends on inputs from $I_j$. We create a separate security policy for each output variable since in many programs the ouputs are available in different environments.

As an example, consider an accounting program, TAX, which accepts a taxpayer identification file, NAME-ADDR, and another file (or set of files) FIN describing the taxpayer's financial situation. TAX produces two outputs: a bill for the taxpayer and a completed U.S. 1040 federal tax form. The taxpayer receives both, but one copy of the bill goes to the author of the TAX program. TAX is to be confined [Lampson] from disclosing to the author of TAX any information private to the taxpayer, thus the bill is to depend only on the name and address of the taxpayer, but not on his financial status. (We assume that all bills for the use of TAX are for the same amount.)

Graphically TAX can be depicted




For this program there are three security policies

$$I_1 = \{\text{NAME-ADDR, FIN}\},$$

$$I_2 = I_3 = \{\text{NAME-ADDR}\}.$$

Thus $y_2$ and $y_3$ are to depend only on the NAME-ADDR file, presumably the name and address of the taxpayer. A sound protection mechanism for TAX and $(I_1, I_2, I_3)$ need only insure that neither copy of the bill depends on information in FIN.

We are defining a theory of security, a subject about which we have many intuitive notions, so it is important to cover the very large class of security policies that one might reasonably wish to enforce using a protection mechanism in a computer system. We distinguish between 'access control' policies which specify whether a particular access operation to manipulate an object containing information is permitted and the more general 'information control' policies which restrict the use of information. For example, enforcing an access control policy which specifies that the operation READFILE(F) cannot be performed on the object F, is not the same as insuring that the information in F is not extracted. There may be a sequence of operations (excluding READFILE(F)) which in effect extracts the information encoded in F. Access control policies do not take into account the semantics of the operations permitted and prohibited, and thus are a proper subset of the information control policies.

To illustrate the difference by an analogy, consider a painter (analogous to a program) who is to paint a picture (the output) without green paint. This could be enforced by an access control protection mechanism in the framework we have already defined, provided all the green paint were in a known set of pots. A sound protection mechanism would simply not allow the painter to dip his brush into those pots.

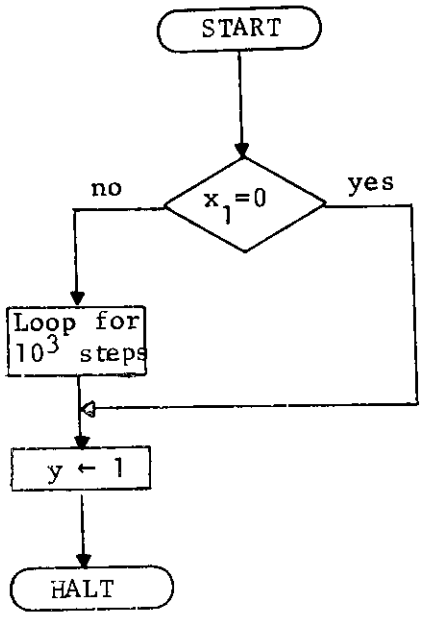
However, the policy of forbidding the use of green paint is not an access control policy; it is an information control policy; for if the painter is simultaneously able to obtain blue and yellow paint, he can mix green paint and use it, violating the original policy to be enforced. Most protection

mechanisms found in extant programmed systems, particularly operating systems, are capable of enforcing some subset of the access control policies and not the more general information control policies. We have not restricted our framework to access control policies, though we could alter it to do so. Instead we consider a subset of the larger class of phenomena -- the information control policies. This should become evident in the presentation of the surveillance protection mechanism in the next section.

The definition of sound is central to our theory. Is it correct? More exactly, does the mathematical definition of sound correspond to our intuitive notion of what it should be? Since this is an extra-mathematical question, no proof can be found. However, we believe that it does correpond to our intuitive notion provided the Observability Postulate is valid. The Observability Postulate states that:

'All observable attributes of a program are actual outputs.'

That is, all ways a program can communicate information to an outside observer are a priori specified. Two examples should help demonstrate the importance of the Observability Postulate. Consider first the case of programs which have as output a single computed value, available outside the program after termination of the program's execution. In this case the Observability Postulate is easily shown to be violated: The following program M that always outputs '1' would appear by our definition to be sound for any security policy:




Note that the argument that M is sound for any policy is based on the fact that M is a constant function. However, we can simply observe the running time of M and conclude whether $x_1=0$ or $x_1\neq 0$. This failure of our notion of sound stems not from any failure in our definitions, but from the fact that the Observability Postulate was violated. M's running time was an implicit output.

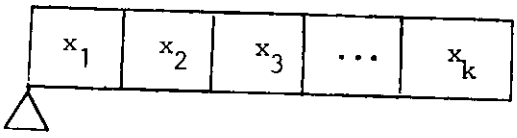
The above example is compatible with our framework as follows: Let us agree that programs will output not just computed output value(s), but a summary of their entire computational history. In this case, they will also output a single variable T (standing for 'time', the only computational history observable) which is the elapsed real time, the elapsed compute time or the number of steps executed. Now, the Observability Postulate is no longer violated. In the above program M is no longer a constant function: the length of its computation depends upon $x_1$; hence, M is not sound for the security policy $I = \{x_2\}$ or $I = \{\}^*$.

*$\{\}$ denotes the empty set.

By the nature of computer systems, there are a set of attributes of computation which may be observed. The pattern of sound waves produced by printer hammers hitting a revolving print chain may be sufficient for an eavesdropper with some sound equipment to detect what is being printed. Heat radiation of computer components may be detectable. The pattern of tape movements can encode information visably detectable by someone in the computer room. Even patterns of a program's use of operating system resources may be detectable by other programs. This, too, constitutes a (potential) output.

Before a sound protection mechanism for a program can be defined, all these observables must be specified as outputs of that program. In the 'worst case', the program will be defined to output $S_0,\ldots,S_m$ where $S_0$ is the initial state of the program, $S_m$ is the final state and $S_{i-1}$ follows directly from $S_i$ for $i=0,\ldots,m-1$. What comprises the state has been determined by what a priori is known to be observable.

One further example should reinforce the subtlety and importance of the Observability Postulate. Let our programs have inputs that are placed on a linear 1-way read only tape (automata theorists read Turing tape; non automata theorists read magnetic tape) with the read head initially at the leftmost character:




Consider a protection mechanism M and the security policy $I = \langle x_2 \rangle$. We assert that M can never both read $x_2$ and also be sound as long as our programs output their entire computational history. This follows since in order for M to get

to the part of the tape where $x_2$ is stored it must move across $x_1$. Even if M does not 'look at $x_1$', it will encode the length of $x_1$ into the computation sequence of M; hence, M will not be sound. Now we see that our definition of sound is quite correct in not allowing M to be sound. However, suppose that we wish to be able to allow inputs to be on linear tapes; how can we avoid this problem? One answer is to add a new operation, say, tab(i). This operation in one step causes the read head to jump directly to the input part of the tape corresponding to $x_i$. Now we can indeed have a protection mechanism that can read $x_2$ and is sound. But a new problem arises: is the Observability Postulate still valid? Perhaps tab(i) takes time dependent on the length of $X_1,\ldots,x_{i-1}$, and moreover perhaps this time is observable. This is the crux of the problem and there seem to be two answers: (1) run tab(i) so that it uses constant time or (2) apply our methods recursively to tab(i).

In general we can always attempt to make the problem of constructing a protection mechanism easier by causing execution attributes to be made non observable, i.e., by limiting the range or even the existence of outputs. For instance the program M in the first example is sound for $I = \langle x_2 \rangle$ or $I = \langle\ \rangle$, if it is run in a batch environment where its running time cannot be determined by the user who submitted the program to be run.

While soundness is the key bridge between protection mechanisms and security policies, the central issue is not just to construct sound protection mechanisms. The protection mechanism that always outputs some fixed violation notice is certainly sound -- it is also useless. (It's equivalent to pulling the computer's plug out of the wall!) We are therefore led to consider the concept of how 'complete' a protection mechanism is.

Definition. Suppose that $M^1$ and $M^2$ are protection mechanisms for the same program Q and policy I. Then $M^1$ is as complete as $M^2$ ($M^1 \geqslant M^2$) provided for all inputs $\bar{a}$,

$$\text{if } M^1(\bar{a}) \neq Q(\bar{a})^*, \text{ then } M^2(\bar{a}) \neq Q(\bar{a})$$

Also $M^1$ is more complete than $M^2$ ($M^1 > M^2$) provided $M^1 \geqslant M^2$ and for some $\bar{a}$, $M^1(\bar{a}) = Q(\bar{a})$ and $M^2(\bar{a}) \neq Q(\bar{a})$.

The relation $\geqslant$ is a partial ordering on the protection mechanisms for a given program and policy. Also, $\geqslant$ is a practically motivated ordering: Consider a single output program with two protection mechanisms $M^1$ and $M^2$. $M^1 \geqslant M^2$ implies that $M^1$ never gives a violation notice when $M^2$ does not. This implies that the utility of $M^1$ is at least as high as that of $M^2$, for one is only interested in getting non-violation notices. Note, however, that if $M^1 \geqslant M^2$ for a multi-output-variable program and both give a violation notice given the same input $\bar{a}$, that violation notice may not be for the same output variable.

We can show how to 'join' two sound protection mechanisms to form a new sound one which is as complete as each of the other two.

Definition. Suppose that $M^1$ and $M^2$ are protection mechanisms for Q. Define $M^1 \cup M^2$ to be the protection mechanism M defined by

$\forall$ input $\bar{a}$

$$M(\bar{a}) = \begin{cases} Q(\bar{a}) & \text{provided } \exists i,\ M^i(\bar{a}) = Q(\bar{a}),\ i \in \{1,2\} \\ M^1(\bar{a}) & \text{otherwise} \end{cases}$$

The key property of union is that if either $M^1(\bar{a})$ or $M^2(\bar{a})$ is the same output

*$M^1(\bar{a}) \neq Q(\bar{a})$ if there is any output $M^1_j(\bar{a}) \in F$.

as $Q(\bar{a})$, then so is the union, $M^1 \cup M^2(\bar{a})$. Otherwise since both $M^1(\bar{a})$ and $M^2(\bar{a})$ include at least one violation notice so does $M^1 \cup M^2(a)$.

Theorem 1. Suppose that $M^1$ and $M^2$ are sound protection mechanisms for program Q and security policy I. Then $M^1 \cup M^2$ is a sound protection mechanism for Q and I. Moreover, $M^1 \cup M^2 \geq M^1$, $M^1 \cup M^2 \geq M^2$, $M^1 \cup M^2 \geq M^2 \cup M^1$ and $M^2 \cup M^1 \geq M^1 \cup M^2$.

Proof. Immediate from the definitions.

We can easily generalize Theorem 1 to show that from the sound protection mechanisms $M^1, M^2, \ldots$ we can form one all encompassing sound protection mechanism $M = M^1 \cup M^2 \cup \ldots$ such that $M \geq M^i$. Indeed it can easily be shown that the sound protection mechanisms form a lattice; however, we shall not need this observation in the sequel.

Theorem 2. For any program Q and security policy I there exists a sound protection mechanism M for Q and I such that M is maximal. That is, for all sound protection mechanisms M' for Q and I, $M \geq M'$.

Proof. Let $M = \{M' \mid M' \text{ sound for Q and I}\}$. Let M then be

$$\bigcup_{N \in M} N.$$

Then as in Theorem 1, M is a sound protection mechanism for Q and I. Clearly, as in Theorem 1, $M \geq N$ for any sound protection mechanism M; hence, M is maximal.

We have established that the maximal protection mechanism always exists; however, as we shall show in Section V, they cannot always be constructed.

Therefore the central problem is: given a program Q and a security policy I, find a sound protection mechanism M that is as 'high' as possible under the completeness ordering. In the next two sections two protection mechanisms are presented. Later in Section V these and other mechanisms are compared under the $\geq$ ordering.

## III. SURVEILLANCE PROTECTION MECHANISM

This section both illustrates a new protection mechanism and the framework developed in the preceding section. In order to define this mechanism we will first restrict our programs to be flowcharts with a single output. We will then show how to assign to each flowchart and security policy a protection mechanism called the surveillance protection mechanism. This protection mechanism is then proved to be sound in Theorem 3.

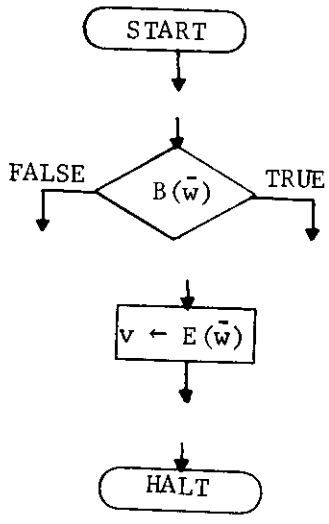
Definition. A flowchart F with input variables $x_1,\ldots,x_k$ program variables $v_1,\ldots,v_m$ and output variable y is a finite connected directed graph whose nodes are boxes of the form:

1. START BOX:

2. DECISION BOX:

3. ASSIGNMENT BOX:

4. HALT BOX:




($B(\bar{w})$ is a predicate)

($E(\bar{w})$ is an expression and v is a program or output variable)

Since we are concerned with executing programs expressed as flowcharts, we must assure ourselves that the Observability Postulate is valid. We will

assume that each flowchart F defines a program Q as follows:

$$Q: \underbrace{\mathbb{N}\times\ldots\times\mathbb{N}}_{\text{k copies}} \rightarrow (\mathbb{N} \times \mathbb{N})$$

where k is the number of input variables of F and $\mathbb{N}$ is the set of natural numbers (i.e. $\mathbb{N} = \{0,1,\ldots\}$).

Suppose now that $\bar{a}$ is an input for Q. Then $Q(\bar{a})$ is a pair, the first component of which is the computed output and the second component of which is the number of steps executed.* We ascribe to the flowchart F the usual semantics: There is exactly one start box; execution begins there by initializing $\bar{x}$ to $\bar{a}$, and $\bar{r}$ and y to $\bar{0}$ and 0. C is defined to be the program counter of F, a variable which contains the name of the next box to be executed. C is initialized to contain the successor of the start box. Execution then follows the logic of the flowchart; at a decision box the path corresponding to the predicate's truth value is taken. The predicates in the decision boxes and the expressions in assignments are assumed to be interpreted in the sense of schemata theory [Manna]; however, no specific assumptions are made about what predicates and expressions are allowed. Note that inputs do not get assigned during a computation.

A halt box is executed by making available to the external world an ordered pair consisting of the value of y and the number of steps executed.

Definition. Suppose that Q is a flowchart program.** Associate with each

---

*Therefore we will be encoding running time. We might have chosen number of page faults, kilowatt-hours consumed, or the sequence of boxes executed. However, we have arbitrarily selected running time as a representative attribute of execution that is observable.

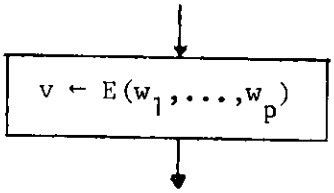
**'flowchart program Q' is a shorthand for 'flowchart that defines the program Q'.

variable v of Q (input, program, output) a new boolean valued variable $\tilde{v}$ called the surveillance variable of v. Suppose further that I is some security policy for Q. Then the surveillance protection mechanism M corresponding to Q and I is the flowchart program that is constructed as follows:

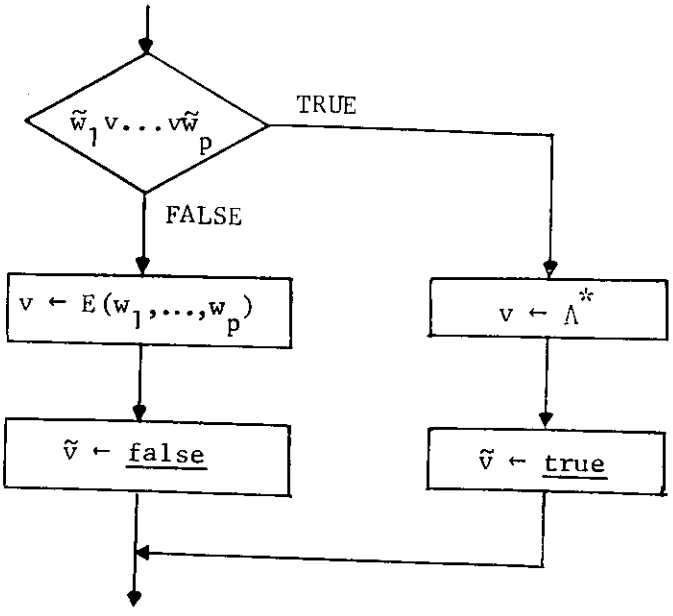
The surveillance variables are program variables of M; the input, program and output variables of Q have the same role in M. M follows from Q by the following transformations:

1. Insert directly after the START box assignments that set $\tilde{v}$ to true if v is an input not in I and false otherwise.

2. Replace the assignment box in Q

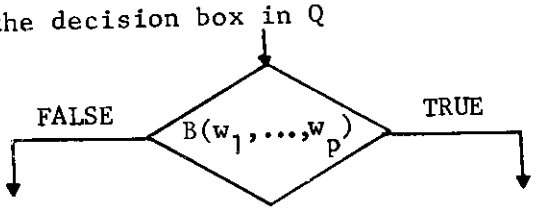



by the following

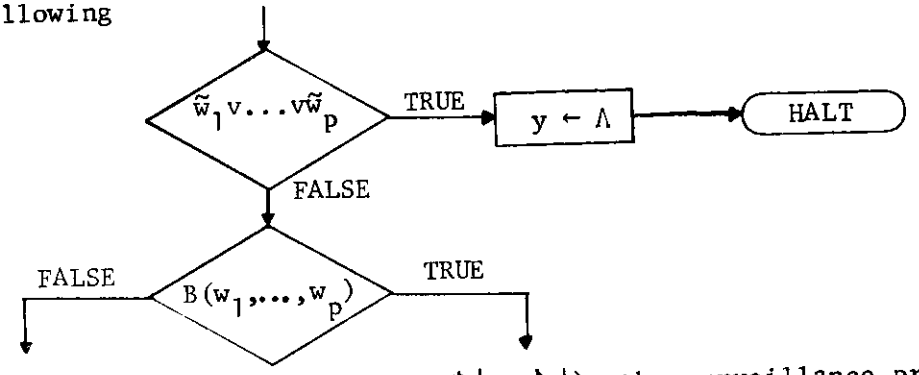



*Λ is a new symbol.

3. Replace the decision box in Q




by the following

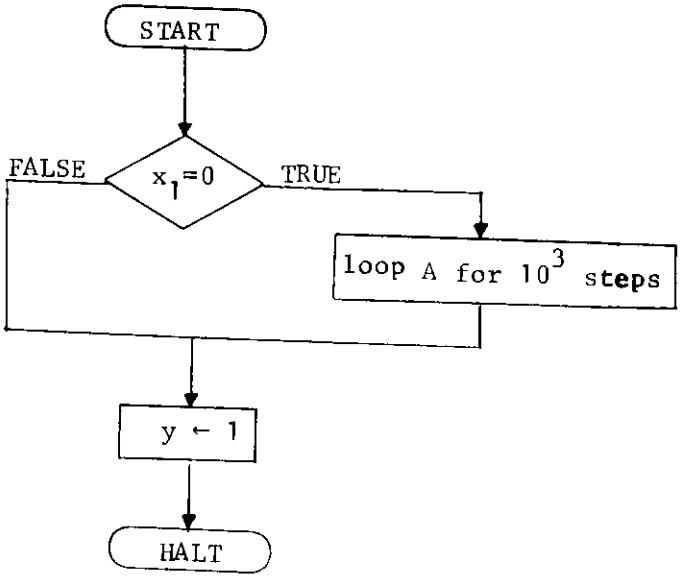



Where a flowchart program Q has a range $(\mathbb{N} \times \mathbb{N})$, the surveillance protection mechanism for that flowchart and any security policy has a range $((\mathbb{N} \cup \{\Lambda\}) \times \mathbb{N})$. The surveillance mechanism can produce a computed output $\Lambda$, which can be called the 'null output'. A violation notice for the surveillance mechanism is of the form $(\Lambda,i)$ where $i \in \mathbb{N}$.

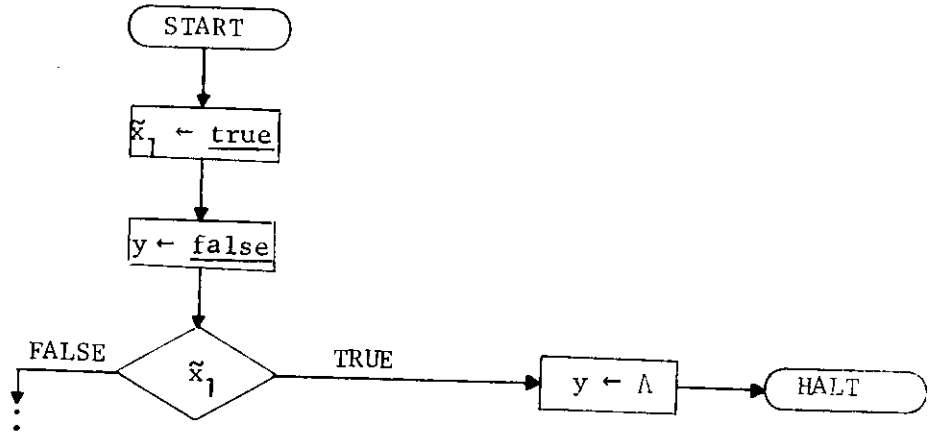
Theorem 3. If M is the surveillance protection mechanism for the flowchart program Q and security policy I, then M is sound for Q and I.

A detailed proof is omitted. However, an easy induction argument shows that no input variable v not in I is ever used in any assignment or decision box that is executed. In order to show that M is sound we need only show it is equal to M' where M' is the same flowchart as M except that all occurrences of v not in I are replaced by some new 'dummy' variable.

In order to illustrate the surveillance protection mechanism, consider the flowchart program $Q^1$:




where A is some computation that loops for some large, but bounded number of steps. Now $Q^1(x_1)$ always halts with y=1; however, $Q^1(x_1) = (1,n)$ where n is the number of steps executed in computing $Q^1(x_1)$ clearly depends on $x_1$. Now let us consider the surveillance protection mechanism $M^1$ for $Q^1$ and the security policy $I = \{\}$.* Then $M^1$ is



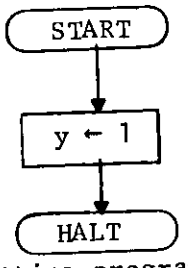

For any $x_1$, $M^1(x_1) = (\Lambda,5)$. That output which is a violation notice indicating that five steps were executed, is independent of $x_1$. Thus the surveillance

*Recall that $\{\}$ is the empty set indicating that the computed output should rely on no input variables.

protection mechanism $M^1$ always outputs the same answer in the same time and there is no way to infer from the output of $M^1$ anything about the value of input $x_1$.

A pertinent question is: how can one implement the surveillance protection mechanism? First, the surveillance protection mechanism could be implemented as part of an interpreter. Since surveillance variables are boolean valued, this would use only a fairly small amount of space, However, the time required would likely be large. Second, the surveillance protection mechanism could be implemented as part of a compiler. One could have a compiler generate either in line code or procedure calls to update and test the surveillance variables as required. It is likely that such an implementation would execute more rapidly than the interpreted version. Third, the surveillance protection mechanism could be implemented in hardware. Each boolean surveillance variable could easily be represented in microcode at the cost of one bit per variable.

Fourth, a new protection mechanism functionally equivalent to the surveillance mechanism could be defined. It would be in the form of a compiler which mapped a program and a security policy into a new program which would be identical to the surveillance mechanism for that program with all computation on surveillance variables evaluated at compile time and thus removed. The example program $Q^1$ would be mapped to




At the cost of possibly generating programs larger than the original flowchart programs the compiler mechanism could allow specification of security policies to be postponed at the time of program invocation.

## IV. HIGH WATER MARK PROTECTION MECHANISM

In the last section the surveillance protection mechanism was presented; here we will present a second protection mechanism called the high water mark protection mechanism. It is essentially the basis for some government, particularly military, information systems; it is also an abstraction of the protection mechanism used in the ADEPT-50 operating system described in [Weissman]. The high water mark protection is included here for two reasons: (i) the fact that we can represent it demonstrates the power of our framework; (ii) it is both sound and unsound! The reason for the latter requires some further explanation that depends on the Observability Postulate.

The high water mark mechanism is sound only in the case that the observables do not include certain attributes of the computational history of programs. If flowchart programs output their history, for example their running time as in Section III, then the high water mark protection mechanism is unsound. Essentially it is unsound since it cannot detect that a program is leaking information by varying the run time (as we will later show).
On the other hand, suppose that we modify the definition of flowchart programs so that they only output their computed output variable values. Then we can prove that the high water mark protection mechanism is sound. It is interesting to note that our framework is general enough to allow a mechanism to be both sound and unsound under varying specifications of observables.

In order to define the high water mark protection mechanism, let Q be a flowchart program and let I be a security policy (for the moment we will not specify whether or not Q outputs its computational history as well as its computed output variable). We further assume that we are given a set F

linearly ordered by $>$ and a function $F_0$ from the variables of the flowchart Q to the set F. We also assume that there exists an element p in F such that for any v in I and any w not in I,

$$F_0(w) > p \geq F_0(v)^*.$$

We intuitively interpret this as follows:

1. F is a set of 'classifications' such as 'top secret', 'secret', 'confidential', 'public' which are ordered so that

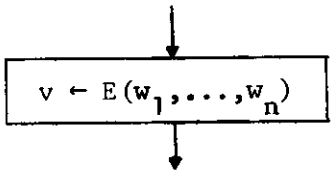
   top secret > secret > confidential > public.

2. Initially all the variables in a security policy I are given some classification equal to or lower than p; all other variables are given some classification higher than p.

As in the construction of the surveillance protection mechanism we associate with each variable v of Q a new variable $\hat{v}$: $\hat{v}$ has values in F. The high water mark protection mechanism M is constructed by the following local transformations on Q:

1. After the START box initialize all variables $\hat{v}$ to $F_0(v)$.

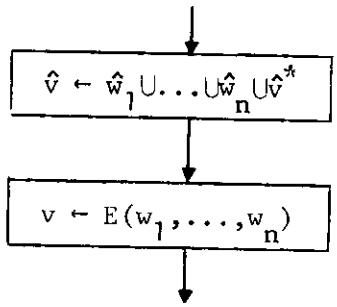
2. Replace the assignment box



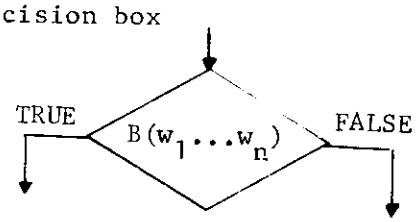

   by

---

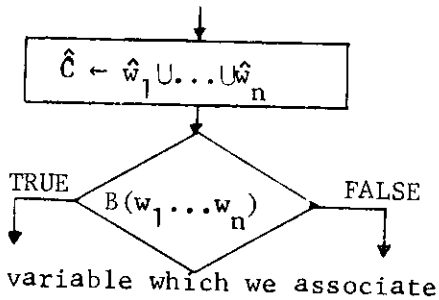
* $a \geq b$ if $a > b$ or $a = b$.




3. Replace the decision box

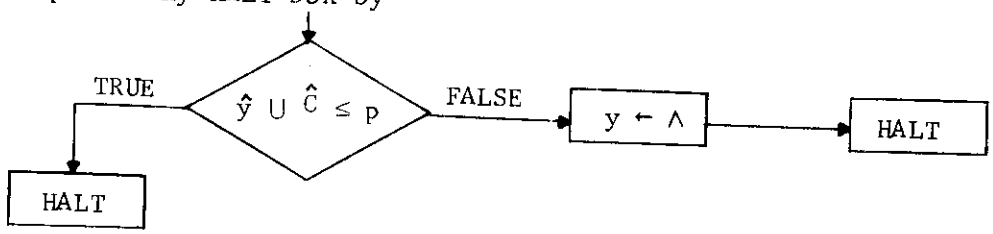



by




where $\hat{C}$ is the variable which we associate with the program counter C.

4. Replace any HALT box by




where $\Lambda$ is the null output value introduced earlier.

Theorem 4. When flowchart programs only output their computed output variables** then the high water mark program mechanism is sound.

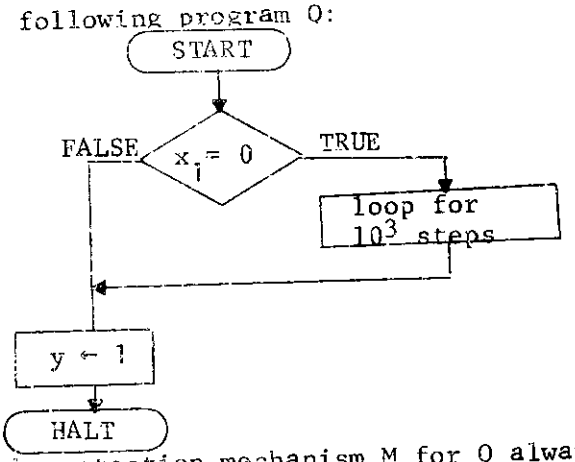
---

* For a,b $\in$ F: a $\cup$ b = maximum under the order $>$.

** By computed output variables we mean those explicitly assigned in the program, not those like running time.

A detailed proof is omitted. The idea of the proof is contained in the name 'high water mark'. The sets $\hat{v}$ simply keep track of all assignments to v. Note in the assignment $v \leftarrow f(w)$, $\hat{v}$ is replaced by $\hat{v}$ and $\hat{w}$; thus, the sets $\hat{v}$ are 'monotone': once an element is placed into them it must remain there. Finally a violation notice (i.e. $\Lambda$) is given if either the output y or the program counter contains any $a \in F$ which is of higher classification than p when the halt box is executed.

In contrast to Theorem 4 we can see that the high water mark protection mechanism is unsound when programs output their running time. It suffices to again consider the following program Q:



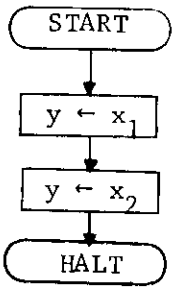

The high water mark protection mechanism M for Q always outputs a violation notice in the form $(\Lambda,i)$. However, the length of the computation sequence of M depends directly on $x_1$: if $x_1 = 0$ it is 4; if $x_1 \neq 0$ it is $10^3 + 4$. Therefore, in this case the high water mark protection mechanism is unsound.

We stated earlier that the Observability Postulate must always be valid -- all potential outputs of a program must be known. This discussion provides one illustration of a protection mechanism which is ineffective or unsound when those observables are of a certain class, i.e. running time.

## V. COMPARISON OF PROTECTION MECHANISMS

In the last two sections the surveillance and high water mark protection mechanisms were presented and both were shown to be sound under the assumption that programs output only their final states. As noted earlier, sound is only part of the story; here their ordering with respect to completeness is considered. While soundness is all or none we will see that with respect to completeness there is indeed a spectrum of possible values.

First let us consider the surveillance and high water mark protection mechanisms. In order to make this comparison meaningful we assume as before that programs only output their computed values. Suppose that $M^S$ is the surveillance protection mechanism for some flowchart Q and security policy I and $M^H$ is the high water mark protection mechanism for Q and I. It is easy to see that $M^S \geq M^H$ is always true. The following simple flowchart and the policy $\langle x_2 \rangle$ shows that $M^S > M^H$ is possible:



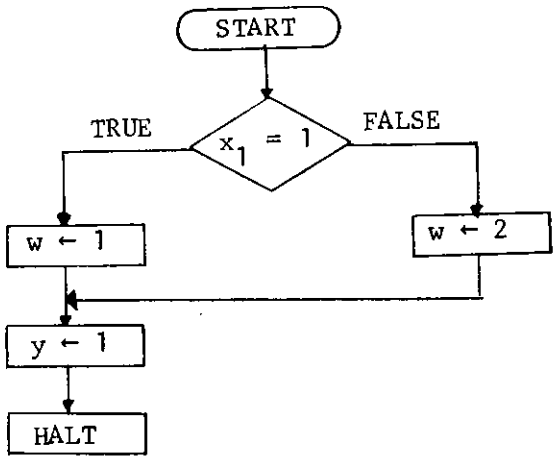

In this case $M^S$ always outputs the value of y, but $M^H$ outputs $\Lambda$. Intuitively surveillance is better here since it allows 'forgetting' while high water mark does not. A more impressive example of forgetting is the following:

> Suppose that during a program Q's computation it must use a block of memory that earlier held 'sensitive data'. Surveillance will allow $Q(\bar{a})$ ($\bar{a}$ = the input value), provided Q correctly 'zeroes' out

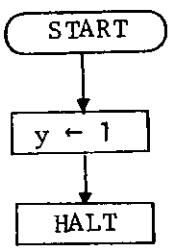
the block of memory; the high water mark mechanism will never output $Q(\bar{a})$, even after this zeroing.

A skeptical reader may reply that this example is artificial. Why not just rewrite Q so that no unneeded assignments are performed? But consider the case of a program where early in the program a variable is assigned a value which is never subsequently used before the variable is reassigned. This is indeed reasonable: often in software that value would have been used had the inputs been different.

While surveillance is more complete than high water mark it is not maximal, i.e. it is not the mechanism that produces the fewest violation notices. In order to see this consider the following program Q:



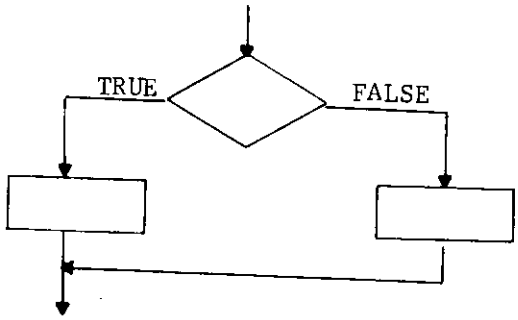

Let $I = \{x_2\}$ be the security policy. Then the surveillance protection mechanism M for Q and I always output $\Lambda$: M never even executes the decision box since $x_1 \notin I$. However, consider the following protection mechanism M' for Q:



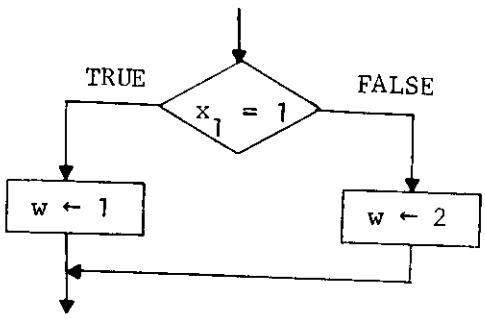
It is easy to see that M' is sound for Q and I. Since M' > M we see that the surveillance protection mechanism is not maximal.

The reason that the surveillance protection mechanism performed poorly on $Q^1$ is that once we branched on $x_1$ there was no way for surveillance to detect that the assignment to y was independent of $x_1$. For the remaining part of this section we will investigate how to modify surveillance so as to make it more complete. We will continue to assume that programs only output explicitly computed values. This is done only for definiteness; whether or not programs also output their running time or any other attribute makes no difference in the following analysis..

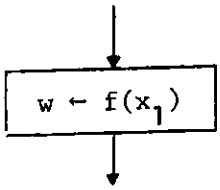
As a step in the direction of improving surveillance suppose that we modified it so that it could detect flowchart occurrences of the form:



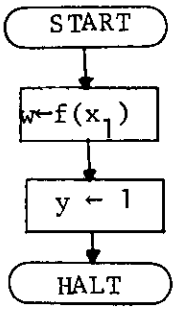

For these '<u>if then else</u>' constructs could we make all future computations independent of whether the <u>then</u> or the <u>else</u> path was taken so that the resulting protection mechanism is still sound? The answer is yes and is demonstrated by looking first at the example <u>if then else</u> of Q:



Clearly, the above is functionally equivalent to

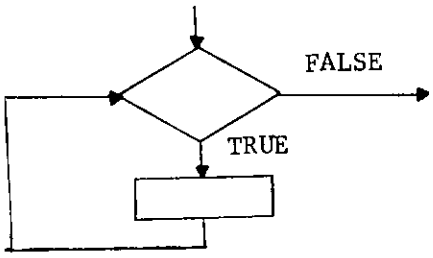



where f(x) = if x = 1 then 1 else 2. Let us transform Q, into Q':



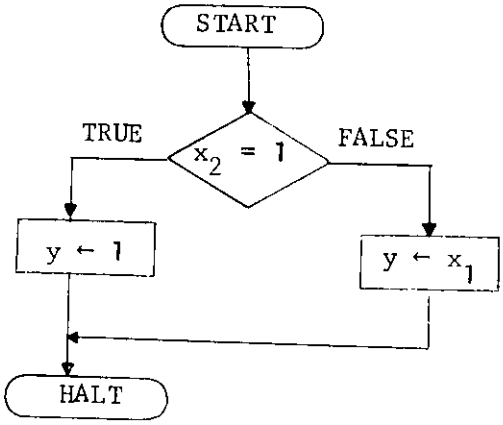

Now the surveillance protection mechanism for Q' and I = $x_2$ always gives the output 1, just as M' did.

This example is just an instance of a general way to generate many different protection mechanisms: Given a program Q transform it to Q' where Q and Q' are functionally equivalent. Then apply the surveillance protection mechanism to Q' to yield a sound protection mechanism for Q. The above 'if then else' transform is but one of many. For instance, we could create a 'for loop' transform that operates on

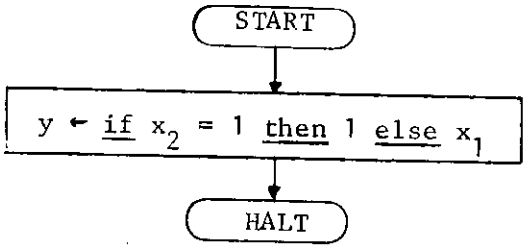



in a way analogous to the if then else transform. Indeed transforms can be created for all single entry and single exit structures.

Is the application of such transformations always advisable? Unfortunately the answer is no. Consider the flowchart program Q:




Let $I = \{x_2\}$ be the security policy again. Let M be the surveillance protection mechanism for Q and I; also let M' be the protection mechanism that corresponds to the if then else transform on Q. Clearly since M' is the surveillance protection mechanism for the following program:

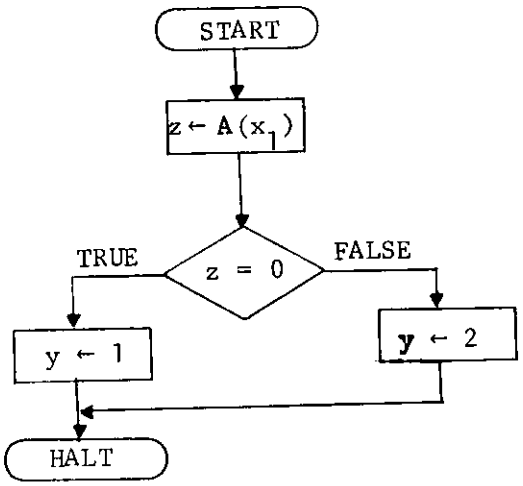



M' always outputs $\Lambda$. On the other hand, M outputs 1 provided $x_2 = 1$; hence, $M \succ M'$. Thus, the danger is that since one does not know which branch was taken one must assume the worst case.

In summary, whether to apply a transform or not is not a clearcut decision. The optimal way to do this is yet unclear. Indeed we can prove that there is no effective way to get the maximal protection mechanism.

Theorem 5. Given a flowchart program Q and a security policy I it is not effectively decidable if M is sound and maximal for Q and I.

Proof. Consider the following program Q and the security policy I that is the empty set:




Let M be the maximal protection mechanism. We assume that A(x) is the result of some arbitrary total function with A(0) = 0. Now clearly M(0) = 1 or $\Lambda$. We now claim that

(1) M(0) = 1 if and only if $\forall x$ A(x) = 0.

Since I is empty, M must be a constant function. If $\forall x$ A(x) = 0, then M always equal to 1 is clearly maximal. Now assume that A(b) $\neq$ 0; we must show that M(0) $\neq$ 1. Clearly, M(b) = 2 or $\Lambda$. In case M(0) = 1, M cannot be constant; hence, (1) is true.

Now (1) shows that if we can effectively construct M, then we can effectively determine whether or not $\forall x$ A(x) = 0; this, however, is impossible. □

The fact that the maximal protection mechanism is not effectively constructible should be contrasted with a similar result from capability theory [Harrison]. Essentially it has been shown that one cannot effectively determine if a program will ever make an 'illegal access'. This results from the fact that it is impossible to <u>a priori</u> determine which execution sequence a program will take. In a sense, therefore, one cannot determine statically

(or at compile-time) whether or not a program will behave properly. On the other hand, Theorem 5 demonstrates that it is impossible to determine whether or not a program has obtained 'illegal information' even if the exact execution sequence is known. (Recall protection mechanisms can be completely dynamic.)

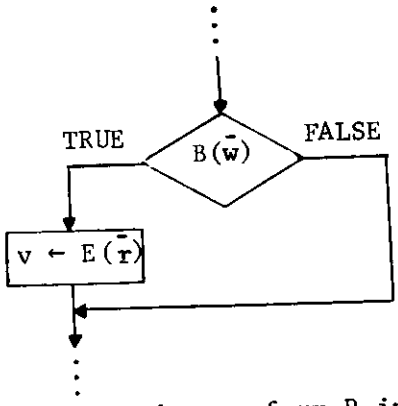
## VI. PROTECTION MECHANISMS EXTENSIONS

The surveillance protection mechanism for flowchart programs is not sufficient for practical computation. If possible, it should be extended to more complex programs: programs that allow procedures, pointer variables, and parallelism are possible candidates. Some of these extensions, as indicated in Section V, will be straightforward. Others, however, may be difficult.

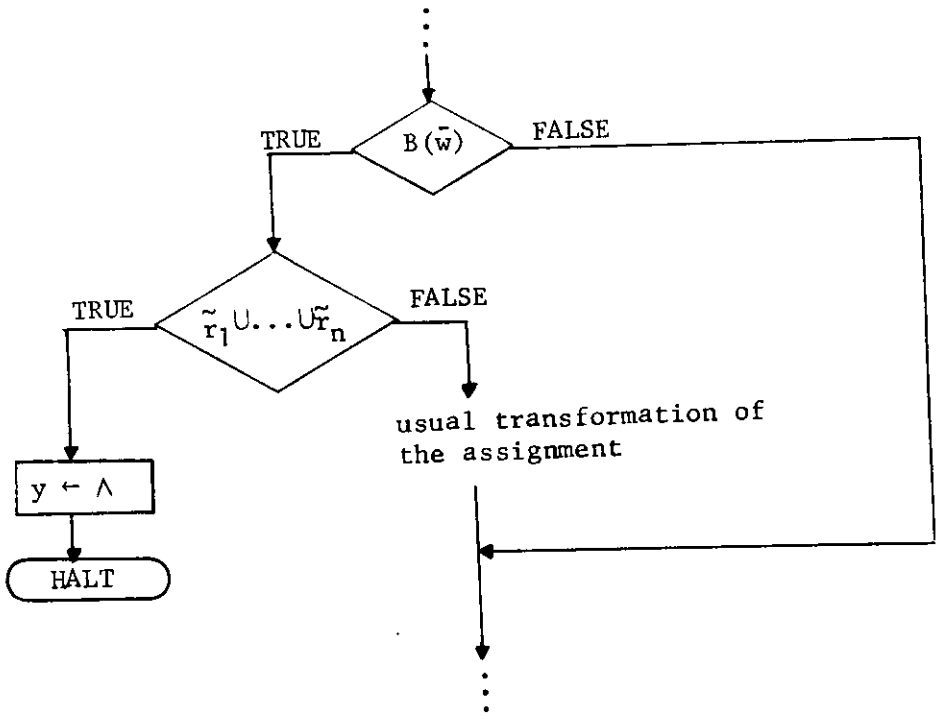
We wish to discuss a simple example of the difficulty of correctly extending the surveillance mechanism. Hopefully this example will show some of the subtleties of the area of security enforcement.

As in Section IV we will for the rest of this section assume that programs only output their explicitly computed outputs. It is under this seemingly simplifying assumption that we will present an example of the difficulty of extensions to the surveillance mechanism.

A reasonable extension to the surveillance method is the following: Suppose that R is a single entry and single exit part of a larger flowchart; for example, R is

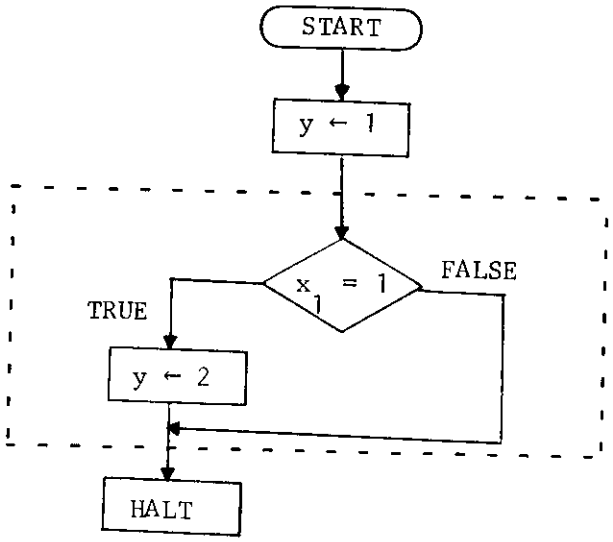



Then let the surveillance method transform R into:




The rationale here is that since R is single entry and single exit soundness can only be violated by executing the assignment; if the assignment is not executed, then there seems to be no way to 'remember' which branch of the decision box was taken. The claim would then be that this method is sound. This method is attractive since it is higher under the $\geqslant$ ordering than the surveillance method. Unfortunately, this extension is not sound.* In order to see this consider the program Q and the security policy $I = \langle x_2 \rangle$ where Q is:

*It is not sound even in the presence of our assumption that only explicitly computed values are output.




The region inside the dotted lines is R. The surveillance protection mechanism for Q and I always outputs $\Lambda$. On the other hand, the extension yields a program that outputs $\Lambda$ if $x_1 = 1$; otherwise, it outputs 1. Clearly, this is not sound.

Intuitively the above unsound extension overlooks what one might call a 'negative inference'. The extension operates fine when we explicitly set y to 2 after testing $x_1$; it fails when we do nothing to y after testing $x_1$. A classic negative inference is due to A. C. Doyle [Doyle].

Holmes: "Then there was the curious incident of the dog in the nighttime."

Watson: "The dog did nothing in the nighttime."

Holmes: "That was the curious incident."

## VII. CONCLUSIONS

The security area currently lacks unity in its basic definitions and terminology. A contribution of this work is the isolation and precise statement of the key questions and concepts needed in any theory of security. It appears to us that the following questions are central to any such theory:

1. What is to be enforced?
2. What is to do the enforcing?
3. Does it do the enforcing?
4. If it does, then how well does it do the enforcing?
5. What assumptions, if any, are made in answer (3)?

These questions are expressed precisely as follows in our theory:

1'. What is the security policy?
2'. What is the protection mechanism?
3'. Is the protection mechanism sound?
4'. How complete is the protection mechanism?
5'. Does the observability postulate hold?

Not only are these the key questions but our framework is general. It is not biased toward any particular solution for providing security. For example it can be used to model capability systems as well as surveillance.

## ACKNOWLEDGMENTS

We would like to thank Bob Chansler for reading several versions of the paper and Stan Eisenstat for several helpful suggestions.

REFERENCES

[Allen] Allen, F., "Control Flow Analysis," Proc. ACM SIGPLAN Symposium on Compiler Optimization, SIGPLAN Notices, Vol. 5, No. 7 (July 1970), 1-19.

[Bell] Bell, D. W., Secure Systems: A Refinement of the Mathematical Model, The Mitre Corporation MTR 2547, Vol. III, April 1974.

[Doyle] Doyle, A. C., "Silver Blades," The Memoirs of Sherlock Holmes, 1874.

[Harrison] Harrison, M. A., W. L. Ruzzo and J. D. Ullman, On Protection in Operating Systems (to be published).

[Jones] Jones, A. K., Protection in Programmed Systems, Ph.D Thesis, Carnegie-Mellon University Technical Report, June 1973.

[Lampson] Lampson, B. W., "A Note on the Confinement Problem," CACM 16, 10 (October 1973).

[Manna] Manna, Z., Mathematical Theory of Computation, McGraw Hill, 1974.

[Popek] Popek, G. and C. S. Kline, "Verifiable Secure Operating System Software," AFIPS (1974 NCC), 145-151.

[Schroeder] Schroeder, M. D., Cooperation of Mutually Suspicious Subsystems in a Computer Utility, Ph.D Thesis, MAC TR-104, Massachusetts Institute of Technology, 1972.

[Walter] Walter, K. G., W. F. Ogden, W. C. Rounds, F. T. Bradshaw, S. R. Ames, D. G. Shumuan, Models for Secure Computer Systems, Case Western Reserve Technical Report 1137, July 1973.

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

[Wulf] Wulf, W. A., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, R. Pollack,"HYDRA: The Kernel of a Multiprocessor Operating System," CACM 17,6 (June 1974), 337-345.