THE ENFORCEMENT OF SECURITY POLICIES FOR COMPUTATION

Anita K. Jones
Carnegie-Mellon University

and

Richard J. Lipton
Yale University

Abstract: Security policies define who may use what information in a computer system. Protection mechanisms are built into a system to enforce security policies. In most systems, however, it is quite unclear what policies a mechanism can or does enforce.

This paper defines security policies and protection mechanisms precisely and bridges the gap between them with the concept of soundness: whether a protection mechanism enforces a policy. Different sound protection mechanisms for the same policy can then be compared. We also show that the "union" of mechanisms for the same program produces a more "complete" mechanism. Although a "maximal" mechanism exists, it cannot necessarily be constructed.

Key words and phrases: completeness, high-water mark, negative inference, observability, protection, protection mechanism, security, security mechanism, sound, surveillance, violation notice

CR categories: 2.11, 4.30, 4.31

1. Introduction

Society differentiates kinds of information and endeavors to control its use. Out of concern for privacy of individuals and the cost of information theft, society has techniques for controlling who can obtain certain information, when they can obtain it, what uses they can put it to, and even who can produce it. We now symbolically represent sensitive information within computer systems. The control of information dissemination has proven to be as difficult to implement in computer systems as in the rest of society.* It is currently the subject of study by many researchers: Bell et al., Denning, Fenton, Jones, Lampson, Neumann et al., Popek, Rotenberg, Schroeder, Walter et al., Weissman, Wulf et al. [1,2,4,5,6,7, 10,11,12,13,14,15].

* Benjamin Franklin observed that "Three may keep a secret, if two of them are dead."

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. Both authors were supported in part by the Rand Summer Institute on Security.

The need for precise and complete understanding of the basic questions in the security area 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 it is long, poorly structured, and based on imprecisely defined terms. We conclude that to be useful the basis of a theory in the security area must be simple.

This paper develops the basic notions relevant to security, providing a framework in which the underlying principles of security can be investigated. We believe it to be precise, yet simple. As we illustrate, a clear understanding of these basic notions makes evident the inadequacies of some of the proposed approaches to providing security.

The basic elements of our theory are precise

definitions of programs, of protection mechanisms, and of security policies. A security policy defines what information is to be protected; it has a non-procedural form. For example, a security policy might state that a user is not to obtain "top secret" information. In contrast, a protection mechanism defines how information is to be protected; it has a procedural form. For example, a protection mechanism might check each operation performed by a user. Soundness defines the bridge between the nonprocedural security policy and the procedural protection mechanism.

2. Basic Model

Security enforcement involves restrictions on the behavior of computer programs. These restrictions involve determining whether or not the output of a program encodes the proper information. This determination clearly depends only on the functional behavior of the program: If two programs are functionally equivalent, then their outputs certainly encode the same information. This observation leads to our view of programs simply as functions from some set of inputs to some set of outputs.

Definition: Define Q to be a program provided

$$Q: D_1 \times \ldots \times D_k \rightarrow E$$

where Q is a total function and D_i is the range of the *i*th input and E is the range of the output.

As demonstrated by Parnas [9] and extended by Neumann et al. [7], programs can have two distinct types of functions. They can be used as "V functions," i.e. they can be used to "view" their "inputs." On the other hand, they can be used as "O operators," i.e. they can be used to "operate" on their inputs. (Here input corresponds to the current state of the system.) Consequently, there are two types of security questions. Suppose that Q is a program, i.e.

$$Q: D_1 \times \ldots \times D_k \rightarrow E.$$

If Q is used as a view function, then the security question is:

> Does the value of $Q(d_1,\ldots,d_k)$ contain any information that it should not?

This is the so-called "confinement" or "memoryless subsystem" question studied by Lampson [6], Schroeder [12], and Fenton [4]. On the other hand, if Q is used as an operator function, then the security question is:

> Has the function Q altered any information that it should not?

This second question has sometimes been called "data security" (Popek [10]). It concerns itself with whether or not information, such as a system table, has been illegally altered and hence lost.

In the rest of this paper we will study the first security question, i.e. the case where Q is used as a view function. We do, however, assert without proof that the same methods used here to study this case can also be used to study the second case. Let us now examine some examples.

Example 1: Fenton [4] studies programs Q of the form

$$Q: D_1 \times \ldots \times D_k \rightarrow E$$

where D_i and E are the set of natural numbers. The value $Q(d_1,\ldots,d_k)$ is the value obtained by the computation of some given Minsky-machine that was started with its *i*th register containing d_1. Fenton studies whether or not $Q(d_1,\ldots,d_k)$ contains in his terms "priv" information. We will return again to this example.

Example 2: Consider a program Q of the form

$$Q: D_1 \times \ldots \times D_k \times F_1 \times \ldots \times F_k \rightarrow E.$$

Here D_i is the set of possible values for the *i*th "directory"; F_i is the set of values for the *i*th "file." The value of $Q(d_1,\ldots,d_k,f_1,\ldots,f_k)$ is the result of some file manipulation program. In this example the *i*th directory will contain information about who can access the *i*th file. We wish, therefore, to know whether or not $Q(d_1,\ldots,d_k,f_1,\ldots,f_k)$ contains any information about a file that was to be denied to us. We will return again to this example.

Our model of security must check carefully that the value of the view function Q is all the information that is available about the inputs. We raise this restriction to the status of a postulate:

The Observability Postulate: The value of $Q(d_1,\ldots,d_k)$ must encode all the information available about the value of $(d_1,\ldots,d_k)$.

Intuitively this statement appears sensible, yet almost too simple and obvious to mention. But it requires careful attention for two reasons. First, it is extra-mathematical in that no proof that it holds can ever be given. Second, there is a series of examples where it does not hold (Lampson [6]). One example of program output that is sometimes overlooked is execution time -- an implicit output that is often available for interactive programs.

Example 1 continued: As Fenton correctly points out, the observability postulate does not hold for his programs. Indeed, both Denning [2] and Fenton leave it open how to handle execution time. One of the contributions of this paper is the understanding of how to handle execution time in a clear manner. This is elaborated later on.

We now turn our attention to the study of protection mechanisms. A protection mechanism acts as a "gatekeeper": It suppresses or alters the output of the program it is protecting. In an operating system the protection mechanism is usually interleaved with the execution of the program being protected. This is not, however,

necessary. One can imagine a static protection mechanism that is implemented at compile time. Therefore, our definition of protection mechanism must be able to accommodate a wide variety of possible types of mechanisms.

Definition: Suppose that $Q: D_1 \times \ldots \times D_k \rightarrow E$ is a program. Then $M: D_1 \times \ldots \times D_k \rightarrow E \cup F$ is a protection mechanism for Q provided for all $(d_1,\ldots,d_k)$ in $D_1 \times \ldots \times D_k$ either

1) $M(d_1,\ldots,d_k) = Q(d_1,\ldots,d_k)$ or

2) $M(d_1,\ldots,d_k)$ is in the set F.

The set F consists of the violation notices of M.

A protection mechanism acts as follows: Suppose that $(d_1,\ldots,d_k)$ is some possible input. Then the protection mechanism can either give $Q(d_1,\ldots,d_k)$ to the user or return a violation notice. The user is to interpret the violation notice as: "It looks as if you (the user) have attempted to view information that is to be denied to you; hence, I (the protection mechanism) am giving you this message."

Example 3: Suppose that $Q: D_1 \times \ldots \times D_k \rightarrow E$ is a program. Then there are two trivial protection mechanisms for Q. The first is the program Q itself. This corresponds, of course, to no protection at all. The second is the program

$$M: D_1 \times \ldots \times D_k \rightarrow E \cup \{\wedge\}$$

where $\wedge$ is not in E and $M(d_1,\ldots,d_k)$ is always equal to $\wedge$. This corresponds to "pulling the plug."

Example 1 continued: In Fenton's model there are two interesting points about his notion of protection mechanisms. First, Fenton does not make clear the distinction between the program Q and the mechanism M. This lack of distinction means, for example, that he cannot compare different types of protection mechanisms. Second, Fenton allows an unusual type of violation notice. In his case the violation notices (the set F) and the possible output of the original program Q (the set E) need not be disjoint. The set F includes the results of partial computations of the program Q. Thus it may be difficult for a user to determine whether or not he is getting $Q(d_1,\ldots,d_k)$; $(d_1,\ldots,d_k)$ is, after all, both the input and a violation notice. Practically speaking, this difficulty may make it particularly hard to find program bugs that cause violation notices.

Example 2 continued: A violation notice in our simple file system might be a message of the form "Illegal access attempted, run aborted."

The purpose of a protection mechanism is to control information; in our framework this leads to the notions of security policy and soundness.

Definition: A security policy I for the program $Q: D_1 \times \ldots \times D_k \rightarrow E$ is a function from $D_1 \times \ldots \times D_k$ to $\mathcal{D}$ where $\mathcal{D}$ is a new set.

The key relation between a protection mechanism and a security policy is whether the mechanism enforces the policy. This relation is called "soundness."

Definition: Suppose that $I: D_1 \times \ldots \times D_k \rightarrow \mathcal{D}$ is a security policy and $M: D_1 \times \ldots \times D_k \rightarrow E \cup F$ is a protection mechanism for the program Q. Then M is sound provided there is a function $M': \mathcal{D} \rightarrow E \cup F$ such that for all $(d_1,\ldots,d_k)$ ($\mathcal{D}$ = a new set)

$$M(d_1,\ldots,d_k) = M'(I(d_1,\ldots,d_k)).$$

A security policy, therefore, acts as an "information filter." A mechanism M is sound provided it behaves as if it received as input not $(d_1,\ldots,d_k)$ but rather $I(d_1,\ldots,d_k)$. The value of $I(d_1,\ldots,d_k)$ has presumably filtered out all the information that was to be denied to the user. Let us now examine some security policies.

1. Suppose that $I(d_1,\ldots,d_k)$ is always 0. Then clearly this security policy is "Allow the user no information."
2. Suppose that $I(d_1,\ldots,d_k)$ is always $(d_1,\ldots,d_k)$. Then this security policy is "Allow the user any information he wants."
3. Suppose that $I(d_1,\ldots,d_k) = (d_{i_1},\ldots,d_{i_m})$. Then this security policy is "Allow the user any information from $d_{i_1},\ldots,d_{i_m}$; deny him all information about the other d_j's."

Each of these security policies is characterized by either allowing information or allowing no information about some input. This type of security policy is the type that is studied in detail here. These policies are the ones most commonly supplied in real systems. For example, after a user logs on to a system, the files in the system can be divided into two classes: those he should be able to access and those he should not. (This partition of files may change with time, but at any instant such a partition exists.) Because of the importance of these policies we introduce a shorthand for them:

Definition: Suppose that $Q: D_1 \times \ldots \times D_k \rightarrow E$ is a program. Let $\underline{allow}(i_1,\ldots,i_m)$ denote the security policy $I: D_1 \times \ldots \times D_k \rightarrow D_{i_1} \times \ldots \times D_{i_m}$ where

$$I(d_1,\ldots,d_k) = (d_{i_1},\ldots,d_{i_m}).$$

Thus the security policy in (1) is $\underline{allow}(\,)$. The security policy in (2) is $\underline{allow}(1,\ldots,k)$. The policy in (3) is $\underline{allow}(i_1,\ldots,i_m)$.

Example 1 continued: Fenton's notions of priv and

null correspond exactly to security policies of the form allow(...).

Our definition of security policy is oriented towards obtaining a simple framework. The definition of security policy as a function, even with the allow shorthand, is not suitable for use in a real system. In such a system much more powerful shorthands are needed. For example, the "*" = all convention used in Multics (Organick [8]). These issues are not addressed here.

We will now present a series of examples. These examples should help in understanding the basic notions of security policy and soundness.

Example 3 continued: Clearly the protection mechanism that always outputs Λ is sound for any security policy. A program as its own protection may or may not be sound.

Example 2 continued: An interesting security policy here is

$$I(d_1,\ldots,d_k,f_1,\ldots,f_k) = (d_1,\ldots,d_k,f_1',\ldots,f_k')$$

where f_i' is f_i if d_i = "YES" and is 0 otherwise. This security policy allows the user information about the i*th* file only in the case that the i*th* directory permits it. Note that the user can always obtain the value of all the directories. Note also that this security policy is not of the form allow(...).

Example 4: Denning [2] and Rotenberg [11] both contain an example of protection mechanisms that leak information via their violation notices. This should not be considered surprising; their examples simply demonstrate unsound protection mechanisms. If M is sound, i.e. $M(d_1,\ldots,d_k) = M'(I(d_1,\ldots,d_k))$, then any decision made by M to output a violation notice can depend only on allowed information.

Example 5: Suppose that we next consider the following "logon" program:

$$Q: D_1 \times D_2 \times D_3 \rightarrow \{\text{true},\text{false}\}$$

where D_1 is the set of userids and D_2 is the set of possible tables that consist of pairs of the form

(userid,password)

and D_3 is the set of passwords. The value $Q(d_1,d_2,d_3)$ is true if and only if (d_1,d_3) is in d_2, i.e. only in the case that the password corresponds correctly to the given userid. Now consider the security policy allow(1,3), i.e. do not let the user have any information from the password table. Then Q as its own protection mechanism (see example 3) is unsound. The reason this program is workable in practice is that the amount of information obtained by the user is "small."

Example 6: The security policies studied here are "information control" policies. They must be distinguished from "access control" policies. For example, enforcing an access control policy that specifies that the operation READFILE(A) cannot be performed is not the same as insuring that information about A is not extracted. There may be a sequence of operations excluding READFILE(A) that has the same effect as READFILE(A).

Example 1 continued: Fenton's protection mechanism is not completely defined. Indeed one reasonable interpretation of it is unsound. The difficulty is the halt statement:

if P = null then halt.

What happens if P ≠ null? One possibility is that in this case an error message (i.e. a violation notice) is output. This is, however, unsound, as the following program demonstrates:

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

This program will output an error message if and only if x = 0. If the security policy wishes to deny information about x, then this is unsound. Intuitively, the difficulty here is what we call "negative inference." When the program behaves correctly (i.e. does not try to halt when it should not) everything is fine; it fails when the program behaves incorrectly. A classic negative inference is due to A. C. Doyle [3]:

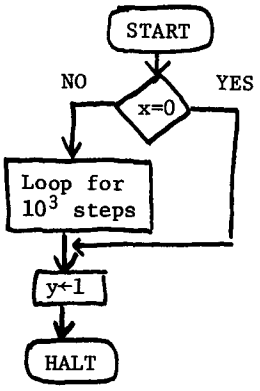
Inspector: Is there any other point to which you would wish to draw my attention?

Holmes: To the curious incident of the dog in the nighttime.

Inspector: The dog did nothing in the nighttime.

Holmes: That was the curious incident.

We next relate the observability postulate and the concept of soundness. Consider the following program Q:



We consider x as the input and y as the output; hence, for any x, Q(x) = 1. For the security policy allow( ), i.e. allow no information about x, the program Q as its own protection mechanism is sound. This follows trivially since Q is a

constant function. We can, however, simply observe the running time of Q to determine whether or not x = 0. This difficulty stems not from the definition of soundness but rather from a violation of the observability postulate: The running time of Q was an implicit output.

This example is compatible with our framework as follows: Let a program's output be not just its computed output value but also its running time, which records the elapsed real time, the elapsed compute time, or the number of steps executed. Now returning to program Q the observability postulate is valid. In particular Q(x) = (1,T), where T is the number of steps executed. Clearly this program is not sound for the policy allow( ). This method is used in section 3 to show how our surveillance protection mechanism can handle running time.

One further example should reinforce the subtlety and importance of the relation between soundness and the observability postulate. Let programs have inputs that are placed on a linear one-way read-only tape with the head initially at the leftmost character:

z_1	...	z_k	...

where each z_i is a block of characters. Consider a security policy allow(2), i.e. allow information only about the second block. Then we claim that no program Q can read z_2 and also be sound, provided running time is observable. This follows since, in order for Q to get to the part of the tape where z_2 is stored, it must move across z_1. Even if Q does not "look" at z_1, it will encode the length of z_1 into the computation of Q; hence, Q will not be sound. 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 *ith* block. Now Q can read z_2 and be sound. But a new problem arises: Is the observability postulate valid? Perhaps tab(i) takes time dependent on the length of $z_1,\ldots,z_{i-1}$? This is the crux of the problem, and there seem to be two answers: Either (1) run tab(i) so that it uses constant time, or (2) apply our methods recursively to tab(i).

While soundness is the most important 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 -- recall example 3. 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 ($M^1 \succ M^2$) provided, for all inputs a (a stands for $(a_1,\ldots,a_k)$), if $M^2(a) = Q(a)$ then $M^1(a) = Q(a)$. Also M^1 is more complete than M^2 ($M^1 \succ M^2$) provided $M^1 \not\succeq M^2$ and, for some a, $M^1(a) = Q(a)$ and $M^2(a) \neq Q(a)$.

The relation $\succ$ is a partial ordering on the protection mechanisms for a given program and policy. Also, $\succ$ is a practically motivated ordering: Consider a single output program with two protection mechanisms M_1 and M_2. $M_1 \succ M_2$ 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 interested only in getting non-violation notices.

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

Definition: Suppose that M_1 and M_2 are protection mechanisms for the program Q. Define $M_1 \vee M_2$ to be the protection mechanism M defined by

$$\forall \text{ input } a,\ M(a) = \begin{cases} Q(a) \text{ provided} \\ \exists i,\ M_i(a) = Q(a),\ i \in \{1,2\} \\ \\ M_1(a) \text{ otherwise} \end{cases}$$

The key property of union is that if either $M_1(a)$ or $M_2(a)$ has the same output as $Q(a)$ then so has the union, $M_1 \vee M_2(a)$. Note that, even though the definition is not symmetric, $M_1 \vee M_2 = M_2 \vee M_1$.

Theorem 1: Suppose that M_1 and M_2 are sound protection mechanisms for program Q and security policy I. Then $M_1 \vee M_2$ is a sound protection mechanism for Q and I. Moreover, $M_1 \vee M_2 \not\prec M_1$ and $M_1 \vee M_2 \not\prec M_2$.

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 define one all-encompassing sound protection mechanism $M = M_1 \vee M_2 \vee \ldots$ such that $M \not\prec M_i$. Indeed it can easily be shown that the sound protection mechanisms form a lattice; we shall not, however, 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, $M \not\prec M'$.

Proof: Let $\mathcal{M} = \{M' \mid M' \text{ sound for Q and I}\}$. Let M then be $\bigcup_{N \in \mathcal{M}} N$. Then as in Theorem 1, $M \not\prec N$ for any sound protection mechanism M; hence, M is maximal. □

(Note that M, while well defined as a function,

may not be recursive -- even if Q is.)

We have established that the maximal protection mechanism exists, but as we shall show later, it cannot always be constructed.

We have now completed the development of our security model. The value of any model is in its useful application. We find that having the model's precise definitions, which are independent of any particular model of computation or any mechanics of protection mechanism implementation, aids in understanding and appraising work in the security area. For example, we have already noted that if all pertinent observables cannot be specified then the "forgotten" observables provide a means for leaking information. Such was the case for the PDP-10 Tenex system where the presence or absence of page faults could be used to obtain passwords.

In summary, we should point out that we have informally used words like "information flow" and "dependence." What these terms mean is precisely captured by the definition of soundness. Though our security model can be easily understood informally, it is mandatory that it be precisely defined as well. Without a precise basis on which to build, one will never be able to make progress in convincing others about the security properties of a system.

3. Surveillance Protection Mechanism

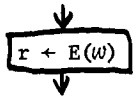
This section both illustrates a new protection mechanism and develops the framework put forth in the preceding section. In order to define this mechanism we will restrict our programs to be flowcharts with a single output and policies of the form allow(...). We will then show how to assign to each flowchart and security policy a protection mechanism called the surveillance protection mechanism. This mechanism and a modified version that ignores runtime are then proved to be sound.

Definition: A flowchart F with input variables $x_1,\ldots,x_k$ and with program variables $r_1,\ldots,r_m$ and with output variable y (i.e. those variables used for temporary values) 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:

Here $B(w)$ is a predicate and $E(w)$ is an expression. We ascribe to the flowchart F the usual semantics: The values of the variables (input, program, and output) are the integers. There is exactly one start box; execution begins there by initializing all the program variables and the output variables to 0. Since we view the flowchart F as computing a program Q with domain $\mathbb{Z} \times \ldots \times \mathbb{Z}$ ($\mathbb{Z}$ = integers), we assume that each input variable is initialized to the value that corresponds to the current input. Thus if $(a_1,\ldots,a_k)$ is the input value, then x_i is initialized to a_i. Execution then follows the logic of the flowchart; at a decision box, the path that corresponds to the predicate's truth value is taken. Please note that no specific assumptions are made about what predicates or expressions are allowed: Any reasonable choice is allowed (i.e. so long as predicates and expressions are recursive there is no difficulty.)

The observability postulate makes it necessary to state exactly what the range of Q (the program computed by F) is. Here we will assert that we will study these flowcharts under two distinct assumptions. First, we will study the case where the range of Q is $\mathbb{Z} \times \mathbb{Z}$. That is, the value of $Q(a)$ has two components. The first is the value of y when the flowchart halts; the second is the number of steps executed by the flowchart, i.e. the "time" used by the flowchart in computing the value of y. Therefore, we will be encoding the running time of our flowcharts. (Note that we might have chosen any of a number of attributes other than running time, but we feel that it is representative.) Second, we will also study the case where the range of Q is just $\mathbb{Z}$. In this case the value of $Q(a)$ has only one component: the value of y when the flowchart execution halts. This second case is the one where we assume that running time is not observable to a user.

We next describe the surveillance protection mechanism. It is based on the idea of keeping track of what inputs a variable may "depend on." Essentially we will associate with each variable in the program a new variable $\underset{\sim}{y}$. The value of $\underset{\sim}{y}$ will be the set of indices of input variables that may have affected the current value of v in some way. A key point here is that we must keep track of $\underset{\sim}{y}$ not only for input, program, and output variables but also for the program counter of the program. The need to do this for the program counter is independently illustrated in Fenton [4]. Therefore, let the program counter of Q be denoted by C.

Definition: Suppose that Q is a flowchart program.† Associate with each variable v of Q (input $x_1,\ldots,x_k$; program $r_1,\ldots,r_m$; output y; program counter C) a new variable $\underset{\sim}{v}$ called the surveillance variable of v. The values of $\underset{\sim}{v}$ are always subsets of $\{1,\ldots,k\}$.

We will now construct the surveillance

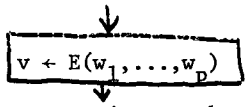
† We will now identify programs with flowcharts.

protection mechanism for the case where time is not part of the output (i.e. when the range of Q is just Z). Then the surveillance protection mechanism M corresponding to the program Q and some security policy I = allow(J) is constructed as follows.

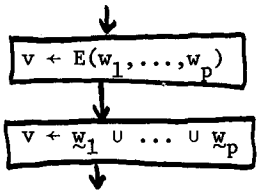
The variables of M are the variables of Q plus the surveillance variables. M is obtained from Q by applying the following transformations:

1) Insert directly after the START box the assignments that set $\underset{\sim}{y}$ to {i} if v is the input x_i and set $\underset{\sim}{y}$ to $\emptyset$ otherwise. ($\emptyset$ is the empty set.)

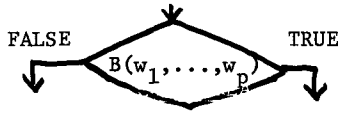
2) Replace each assignment box in Q



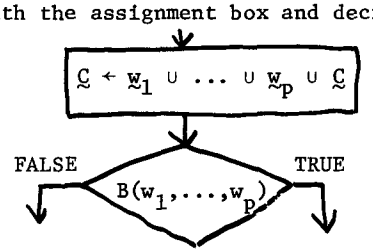
with the two assignment boxes



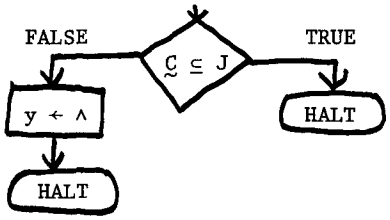
3) Replace each decision box in Q



with the assignment box and decision box



4) Finally, replace each HALT box with



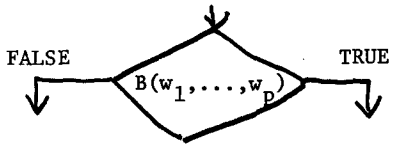
Here ∧ is a symbol that is not a normal output of Q; it is used as a violation notice. Recall that I = allow(J). Clearly, given a program Q and a security policy I, M is indeed a protection mechanism. Indeed, M is always sound:

Theorem 3: If M is the surveillance protection mechanism for the program Q and the security policy I, then M is sound for Q and I provided running times are not observable.

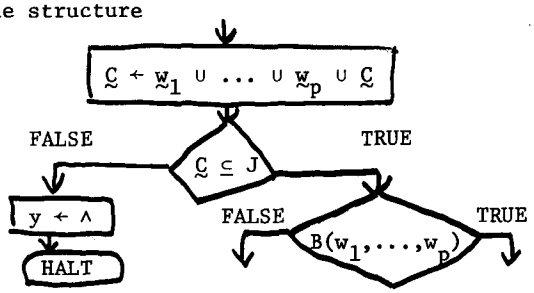
In contrast, it is easy to see that M is unsound when running time is observable. Indeed, both Fenton [4] and Denning [2] point out that it is an interesting problem of how to handle running time. We will now show how our surveillance protection mechanism can be modified to handle running time and still remain sound.

Definition: Suppose that Q is a program and I is a security policy. Then construct the surveillance protection mechanism M' for the case where running time is observable as follows:

Transform Q into M' by replacing each decision box of Q



with the structure



and performing the other transformations as before. Intuitively, if a disallowed variable is about to be tested, flowchart execution is halted and a violation notice is given -- immediately. Then it is easy to see that M' is a protection mechanism. Indeed,

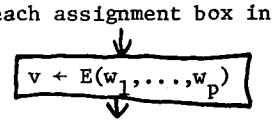
Theorem 3': If M' is the surveillance protection mechanism for the program Q and the security policy I, then M' is sound for Q and I even if running times are observable.

4. Comparison of Protection Mechanisms

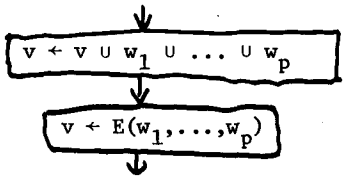
In the last section, the surveillance protection mechanism was presented. It is sound under the assumption that running time is not observable (Theorem 3). As noted earlier, soundness is only part of the story; in this section we consider the comparison of protection mechanisms. While soundness is all or none, completeness provides a measure with which to order these mechanisms.

A possible alternative protection mechanism is the high-water mark protection mechanism. It is related to the mechanism in Weissman [14]. It is based on the same transformations as we used in constructing the surveillance mechanism except that (2) is changed to:

2') Replace each assignment box in Q



with the two assignment boxes

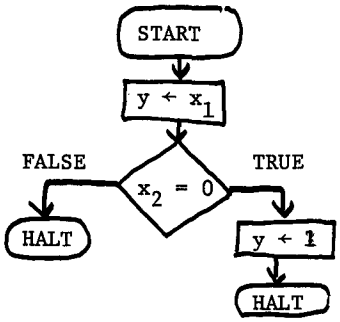


Intuitively, the high-water mark mechanism is based on the assumption that there exists an ordered sequence of security classifications, e.g.

public < confidential < secret < $\overset{\text{top}}{\text{secret}}$.

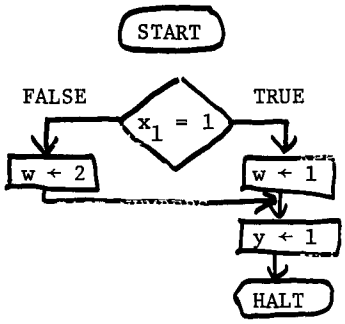
The classification of each variable is initially known and is recorded in an associated surveillance variable. If a variable ever assumes a classification, its future classification is always at least as high.

We will now compare these two mechanisms. Suppose that M_s is the surveillance protection mechanism for some program Q and security policy I and M_h is the high-water protection mechanism for Q and I. It is easy to see that $M_s \not< M_h$, i.e. M_s is at least as complete as M_h. The following simple flowchart and the policy allow(2) shows that $M_s > M_h$ is possible:



In this case M_h always outputs $\wedge$; on the other hand, M_s outputs $\wedge$ only when $x_2 \neq 0$. Intuitively, surveillance is better here, since it allows "forgetting" while high-water mark does not.

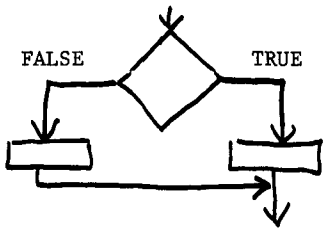
While surveillance is more complete than high-water mark, it is not maximal, i.e. it is not the mechanism that produces the fewest violation notices. In order to see this, consider the following program Q:



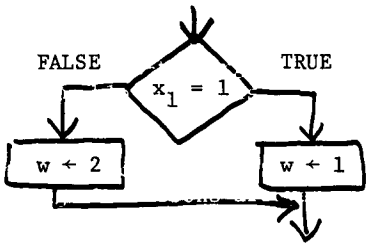
Let I = allow(2) be the security policy. Then the surveillance protection mechanism M_s for Q and I always outputs $\wedge$. Consider, however, the protection mechanism $M_{max} = Q$. It is easy to see that M_{max} is sound for Q_{max} and I. Since $M_{max} > M$ we see that the surveillance protection mechanism is not maximal.

The reason that the surveillance protection mechanism performed poorly on Q is that, once the branch on x there was taken, the surveillance mechanism was unable 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 do not have their running times as observables.)

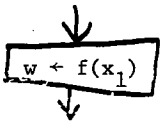
As a step in this direction, suppose that we modify surveillance so that it can detect flowchart occurrences of the form:



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

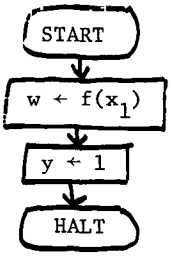


Certainly, the structure above is functionally equivalent to



where $f(1) = 1$ if $x_1 = 1$ and $f(x_1) = 2$ otherwise.

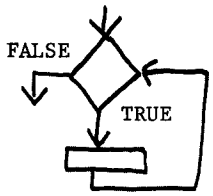
Let us transform Q into Q':



Now the surveillance protection mechanism for Q' and I = allow(2) always gives the output 1; clearly it is maximal.

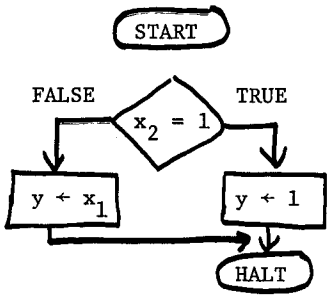
This example is just an instance of a general way to generate many different protection mechanisms: Given a program Q, transform it to Q' where Q and Q' are functionally equivalent.

Then apply the surveillance protection mechanism to Q' to yield a sound protection mechanism for Q. The if then else transform above is but one of many. For instance, we could create a for loop transform that operates on

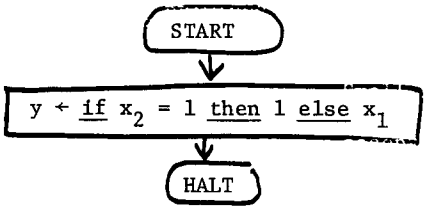


in a way analogous to the if then else transform. Indeed, transforms can be created for all single-entry and single-exit structures.

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



Let I = allow(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 using the if then else transform on Q. M' is the surveillance protection mechanism for the following program:

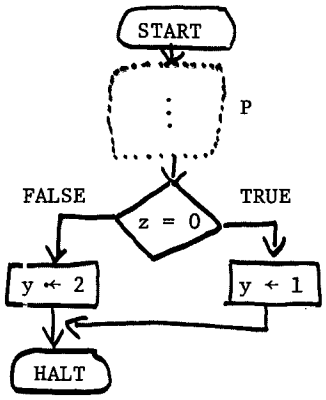


M' always outputs $\wedge$. On the other hand, M outputs 1 provided $x_2 = 1$; hence, $M \not\vdash M'$. The danger is that since one does not know which branch is to be taken one must assume the worst case.

In summary, whether to apply a transform or not is seldom a clearcut decision. But the optimal strategy for deciding is not, as the next theorem shows, computable.

Theorem 4: There is no effective procedure that given a program Q and security policy I outputs a maximal sound protection mechanism.

Proof: Consider the following program Q and the security policy I = allow( ) (i.e. none of the potential inputs are allowed:



Here P is a flowchart fragment that assigns the value $A(x_1)$ to z where $A(x_1)$ is some arbitrary total function such that A(0) = 0. Let M be a protection mechanism. Now M(0) = 1 or $\wedge$. We now claim that

(*) 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 is always equal to 1. Now assume that $A(b) \neq 0$; we must show that $M(0) \neq 1$. Clearly, M(b) = 2 or $\wedge$. If M(0) = 1, then M cannot be constant; hence (*) is true.

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

5. 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 to question 3?

These questions are expressed precisely in our theory as follows:

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.

Acknowledgements

We would like to thank Bob Chansler for reading

several earlier versions of this paper and Stan Eisenstat for a number of helpful suggestions. We are grateful for many useful comments from the referees and our colleagues, especially John Bruno, Stockton Gaines, Mike Harrison, Butler Lampson, Peter Neumann, Jerry Popek, Jerry Saltzer, and Mike Schroeder.

References

[1] D. W. Bell. Secure systems: A refinement of the mathematical model. The Mitre Corporation MTR 2547, Volume III, 1974.

[2] D. Denning. Secure information flow in computer systems. PhD thesis, Purdue University CSD-TR-145.

[3] A. C. Doyle. Silver blaze. The Memoirs of Sherlock Holmes, 1874.

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

[5] A. K. Jones. Protection in programmed systems. PhD thesis, Carnegie-Mellon University, 1973.

[6] B. W. Lampson. A note on the confinement problem. CACM 16(10)m 1973.

[7] P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena. A provably secure operating system. SRI Final Report, 1975.

[8] E. Organick. The Multics System: An Examination of Its Structure. MIT Press, 1972.

[9] D. Parnas. A technique for software module specification, with examples. CACM 15: 330-336, 1972.

[10] G. Popek and C. S. Kline. Verifiable secure operating system software. AFIPS National Computer Conference Proceedings, 145-151, 1974.

[11] L. Rotenberg. Making computers keep secrets. MIT-TR-115, 1974.

[12] M. D. Schroeder. Cooperation of mutually suspicious subsystems in a computer utility. PhD thesis, MAC TR-104, Massachusetts Institute of Technology, 1972.

[13] K. G. Walter, W. F. Ogden, W. C. Rounds, F. T. Bradshaw, S. R. Ames, and D. G. Schuman. Models for secure computer systems. Case Western Reserve Technical Report 1137, 1973.

[14] C. Weissman. Security controls in the ADEPT-50 time sharing system. AFIPS FJCC, 119-133, 1969.

[15] W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and R. Pollack. HYDRA: The kernel of a multiprocessor operating system. CACM 17(6):337-345, 1974.